UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

1

AD-A196 318

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 88-40	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A CLASSIFICATION OF DESIGNATED LOGIC SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED MS THESIS
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) KELLY ANN SHAW		8. CONTRACT OR GRANT NUMBER(s)
PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: SYRACUSE UNIVERSITY		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 1988
		13. NUMBER OF PAGES 133
MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) AFIT/NR Wright-Patterson AFB OH 45433-6583		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE

DTIC ELECTE AUG 0 3 1988 S D

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

SAME AS REPORT

18. SUPPLEMENTARY NOTES

Approved for Public Release: IAW AFR 190-1
LYNN E. WOLAVER [illegible] 18 July 88
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

88

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT

Logic systems can be defined using a trichotomized rather than the dichotomized universe used in classical logic. We partition these designated logic systems using the negation (or complement) function into the designated, antidesignated and neutral logic classes.

The complement function alone is not sufficient to uniquely determine an assignment of logic elements to partition classes. Using a modified definition of conjunction within the designated logic system framework we can uniquely determine an assignment up to equivalence for any designated logic system.

We build a hierarchy of mathematical systems to describe properties found in some common designated Computer logic systems. Starting with a simple algebra and imposing constraints we end up with an order defined for some of the designated logic systems. Using this hierarchy we find the number of conjunction functions homomorphic to classical logic when only designated and antidesignated values are conjoined. The logic systems described by these twelve homomorphisms are designated logic systems closest in behavior to classical logic. (KR)

A Classification of Designated Logic Systems

By

Kelly Ann Shaw
B.S. June 1983, Syracuse University

A Thesis submitted to

The Faculty of

The Graduate School of Engineering and Applied Science
of The George Washington University in partial satisfaction
of the requirements for the degree of Masters of Science

February, 1988

Thesis directed by
Oscar N. Garcia
Professor of Electrical Engineering and Computer Science

TABLE OF CONTENTS

PREFACE

When Dr. Garcia and I began research on this paper over a year ago we started with the idea of using non-classical logic systems in a concrete Computer Science application. We were motivated towards the study of non-classical logic by the research of Mr. Moussavi. He found that classical logic, because it lacked the capability of dealing with unknown quantities, was inadequate to the task of modeling rule based systems. This gave us a concrete example of the usefulness of non-classical logic systems.

As we began to study the topic, however, we found that some of our most basic questions were not answered in the literature. Some of these questions were:

'What is a rule of inference?'
'What is a truth value?'
'What differentiates a logic system from an algebra?'
'What characterizes a logic system?'

As we searched for answers the nature of our research changed. We were no longer searching for an application, we were searching for the nature and definition of a logic system with the hope of working with these systems in a mathematical (or at least non-philosophic) mode.

This paper is the fruit of our research. Although our results can't as yet be used in any concrete practical application, we think they provides a sound basis for understanding non-classical logic systems.

I would like to thank the Air Force Institute of Technology for funding this research.

ILLUSTRATIONS

LIST OF TABLES

ABSTRACT

Logic systems can be defined using a trichotomized rather than the dichotomized universe used in classical logic. We partition these designated logic systems using the negation (or complement) function into the designated, antidesignated and neutral logic classes.

The complement function alone is not sufficient to uniquely determine an assignment of logic elements to partition classes. Using a modified definition of conjunction within the designated logic system framework we can uniquely determine an assignment up to equivalence for any designated logic system.

We build a hierarchy of mathematical systems to describe properties found in some common designated logic systems. Starting with a simple algebra and imposing constraints we end up with an order defined for some of the designated logic systems. Using this hierarchy we find the number of conjunction functions homomorphic to classical logic when only designated and antidesignated values are conjoined. The logic systems described by these twelve homomorphisms are designated logic systems closest in behavior to classical logic.

CHAPTER 1

INTRODUCTION

When thinking of 'logic' we tend to think in Aristotelian terms, or in terms of the Principia Mathematica [Russell and Whitehead, 1910]. The world becomes dichotomized into 'that which is true' and 'that which is false.' There are, however, an infinite number of non-classical logics which lend themselves to real world problems. For example, the future contingency statement "there will be a sea battle tomorrow" would have the value 'unknown' in some non-classical logics since its value cannot be defined without the use of modal operators in an extension of classical logic.

We would like to be able to answer two questions about any Computer Science application using a non-classical logic system: 'what kind of logic do we have,' and more importantly 'what kind of logic do we need?' To do this we must first understand the structure of non-classical logics and find how they are both similar to, and different from classical logic. In other words, we must classify them.

In chapter two we introduce designated and antidesignated logic values and define Designated Logic Systems giving some examples of how some of the deviant logics fit into this framework.

In chapter three we partition the designated logics into three classes (designated, antidesignated and neutral) using the negation (complement) function. Using the cycles of a permutation there are a finite number of classes for any finite logic system enumerated by a regular generating function (the same function used to count integer partitions).

In chapter four we use the logic's conjunction to uniquely determine an assignment of logic elements to partition classes up to equivalence.

In chapter five we build a hierarchy of mathematical systems which characterize properties of the designated logic systems. We use this hierarchy in chapter six to examine a special class of Designated Logic Systems whose behavior is homomorphic to classical logic when conjoining designated and antidesignated values. The twelve resulting conjunction functions are closest to classical logic of all the designated logic systems when the neutral class is not empty.

Neither permutations, classifications nor mathematical hierarchies are new. We hope their use to classify and partition Designated Logic Systems illuminates the nature of these systems and leads to a better understanding of their possible uses in applications of Computer Science.

CHAPTER 2

DESIGNATED LOGIC SYSTEMS

Usually, when we think of logic systems, we think of two valued classical logic. There are, however, an infinite number of logic systems which extend or modify classical logic. In this chapter we will introduce several non-classical logics and define terms used to describe Designated Logic Systems.

2.1 Modal and Deviant Logics

Non-classical logics fall into two categories: modal logics and deviant logics [Haack, 1974]. While modal logics extend classical logic, deviant logics deny some principle of classical logic.

Susan Haack, in her Philosophy of Logics [Haack, 1978], defines modal logic systems as those logics sharing the same vocabulary and valid formulae as classical logic but having additional quantifiers and inferences valid in its system. The set of theorems in classical logic is a subset of the set of theorems in a modal logic. Two modalities she discusses in detail are 'necessary' and 'contingent' formulae. For example, in classical logic proposition p may be either true or false. When preceded by the 'necessary' modal, as in 'necessarily p,' proposition p is always true [Haack, 1978].

Paraphrasing from Rescher, modal logic results when propositional logic formulae are subject to a general form of quantification [Rescher 1968]. Rescher does not limit quantification to the traditional existential and universal quantifiers we normally associate with the predicate calculus. For Rescher, a modal can be any quantifier used in the form '<modal operator><logic formula>' as in the formula "It is necessary that p => p."

Aristotle formulated the two classical modalities, apodeictic (necessarily) and problematic (possibly) [Aristotle, Prior Analytics], but other modal quantifiers exist in, for example, the epistemic (I know x or I think x) and deontic (I should do x) logics.

While modal logics extend classical logic by means of quantification, deviant logics rival classical logic. Haack defines a logic L1 as deviant if "...the class of wff [well formed formulae] of L1 and the class of wff of [Classical Logic] coincide, but the class of theorems/valid inference of L1 differs from the class of theorems/valid inferences of [Classical Logic]" [Haack, 1974].

By her definition, the set of theorems in classical logic is not a subset of the set of theorems in a deviant logic. This is the difference between a modal logic and a deviant logic. In this paper we will be concerned with the deviant logics and classical logic rather than modal logic.

2.2 Truth Values and Rules of Inference.

In any logic system there will be at least two truth values and at least one rule of inference used to manipulate truth values. These two structures characterize all logic systems and are what differentiate logics from algebras. In classical logic we use truth values 'true' and 'false' along with several rules of inference such as modus ponens, modus tollens and resolution.

Therefore, to understand logic systems, we must first understand truth values and rules of inference. Truth values are simply the set of elements in a logic which can be assigned as the value of any logic formula. As we stated earlier, in classical logic the set of logic values is {true,false}.

The definition of a rule of inference depends on the definition of tautology. In classical logic, well formed formulae may take a value which is always true, always false, or sometimes true and sometimes false depending on value assignments to their atomic parts. These well formed formulae are defined as tautologies, contradictions and contingencies respectively. Langer says of a principle of inference "... if a proposition may be asserted (i.e. is "granted" or otherwise "known as true"), and this proposition implies another proposition, then the latter may also be asserted" [Langer, 1953]. Irving Copi states that a tautology is true by its form alone independent of empirical investigation [Copi, 1972]. We then accept the definition of a rule of inference as a tautology of the form '(p) => (q)' where p and q are well formed formulae. Using rules of inference together with postulates of a logic system, we can derive other tautologies in the form of deduced theorems.

2.3 Other Functions in Logic Systems.

Thus far we have discussed only the implication function in logic systems (we will represent the implication function by the infix operator =>). There are other functions usually found in logic systems such as negation (represented by the prefix operator ~), conjunction (represented by the infix operator &), disjunction (represented by the infix operator +) and equivalence (represented by the infix operator <=>).

The negation function is usually a permutation of the logic values where ~x (read "not x") denotes the converse of some formula x. In classical logic, not true is false while not false is true.

Conjunction and disjunction are functions representing the truth value of a series of well formed formulae. In classical logic conjunction of a number of formulae takes the value of false if any one of the formulae is false, otherwise it takes the value true. Disjunction takes the value true if at least one formula in the series is true.

The equivalence function, in classical logic, yields a truth value representing the equality of two well formed formulae. In classical logic, (p) <=> (q) if and only if p and q both have the value true, or both have the value false. Therefore, true is equivalent to true and false is equivalent to false, but true is not equivalent to false.

In classical logic, and in some deviant logics such as Kleene's three-valued logic system and Post logic systems, implication and equivalence functions are defined in terms of conjunction, disjunction and negation. Implication is usually defined as (p) => (q) = ~(p) + (q) and equivalence is usually defined as (p) <=> (q) = ((p) => (q)) & ((q) => (p)). In our discussion of designated logic systems we will classify the logics using the negation and conjunction functions.

2.4 Designated and Antidesignated Truth Values

Logic system with more than two truth values may have definitions of tautology, contradiction and contingent formulae different from their definitions in classical logic. For example, in a logic system with two logic values representing truth, say t1 and t2, a tautology could be defined as a well formed formula taking only the values t1 or t2 for all assignments to its atomic parts.

Rescher defines the terms designated and antidesignated to mean those values in a logic which represent 'trueness' and 'falseness'[4] [Rescher, 1968]. Using these definitions, a tautology is a well formed formula which takes only designated values for all possible assignments to its atomic parts. Likewise, a contradiction takes only antidesignated values [Haack, 1978]. Contingencies will take both designated and antidesignated values depending on the particular substitution instance of values to the formula's atomic components.

Example 2-1. Rescher defines the grouping of designated and antidesignated values in Lukasiewicz' three valued logic (L3) such that all tautologies and contradictions in classical logic hold in L3 as well. In his example, T and I are designated while F and I are antidesignated [Rescher, 1968]. Note that I is both designated and antidesignated. In classical logic the principle of Excluded Middle states that (p) + ~(p) is a tautology. Table 1 shows that Excluded Middle is a tautology in L3 when both T and I are designated (we refer the reader to Appendix 1 for a complete definition of L3).

Table 1

The Principle of Excluded middle in L3.

~	
T	F
I	I
F	T

+	T	I	F
T	T	T	T
I	T	I	I
F	T	I	F

X	X + ~X
T	T
I	I
F	T

Other assignments of values in L3 to the designated and antidesignated categories are possible (<u>e.g.</u> T is designated and both I and F are antidesignated); however, some contradictions and tautologies in classical logic would no longer hold in L3 (e.g. q + ˜q would not be a tautology) [Rescher, 1968].

2.5 Designated Logic Systems.

Rescher's motivation for assigning values to designated and antidesignated sets rests with his desire to force deviant logics to resemble classical logics in their tautologies. One could choose to require the Principle of Excluded Middle to be a tautology in a deviant logic and then assign values to designated and antidesignated sets. Likewise one could choose to hold the Principle of Contradiction (not both p and ˜p) invariant and assign logic values as required. We choose to make as many tautologies hold as possible with the fewest number of restrictions. To do this we will hold the principle of contradiction as an invariant and use Rescher's classification scheme with suitable modifications.

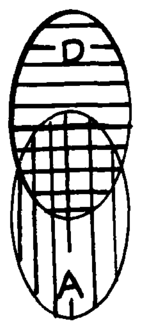
Rescher states that for the principle of contradiction to hold in a logic system all logic values must fall into the designated class or the antidesignated class of logic values. He uses 'or' in its inclusive meaning since a value may be both designated and antidesignated. [Rescher, 1968]. He also constrains the assignments based on negation using the following rules:

> If x is designated then ~x is antidesignated
> If x is antidesignated then ~x is designated

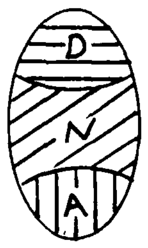
If we call the set of designated values D and the set of antidesignated values A, a Designated Logic System based on Rescher's rules would look like figure 1 (a).

We propose a conceptually similar yet more explicit classification of logic values within a logic system. Let D be the set of strictly designated logic values and A be the set of strictly antidesignated logic values in a logic system. An element x of a logic system is strictly designated if x is designated and ~x is not designated (_i.e._ x is not both designated and antidesignated). An element x in a logic system is strictly antidesignated if x is antidesignated and ~x is not antidesignated (_i.e._ x is not both designated and antidesignated). All logic values not falling into either category will be placed in set N for logically neutral values. This enables us to partition the set of logic values into disjoint subsets as in figure 1 (b). It also allows us to reformulate the rules by which the subsets of logic values are defined. The new rules are as follows:

If x is designated then ~x is antidesignated
If x is antidesignated then ~x is designated
if x is neutral then ~x is neutral.



(a)



(b)

Figure 1. Two Classifications of Designated Logic Systems.

We call logic systems using this partitioning scheme Designated Logic Systems. The remainder of this paper is limited to the investigation and classification of these systems.

2.6 Examples.

Example 2-2. Given a designated logic system of two logic values, the partition of its logic values must fall into one of the two possible partitions shown in figure 2. Figure 2 (a) is the partition in classical logic.

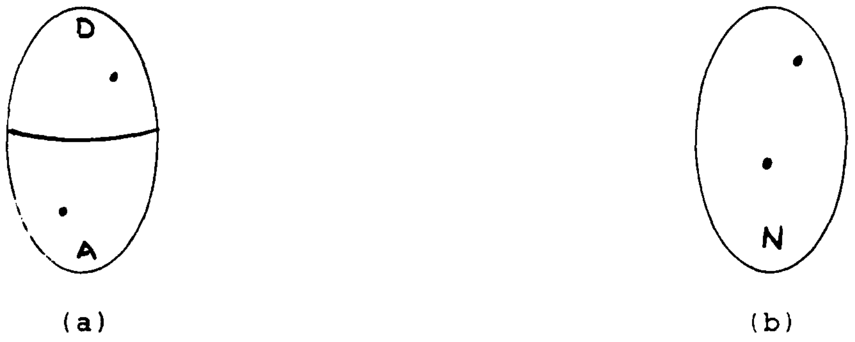


Figure 2. Two Partitions of a Two-valued Logic System.

Example 2-3. In a three valued designated logic system there are also only two possible partitions of the logic values, illustrated in figure 3. Figure 3 (a) represents the partition in Kleene's, Lukasiewicz' and Bochvar's logics while figure 3 (b) represents the partition in Post's three valued logic (complete definitions of these logic systems may be found in Appendix 1).

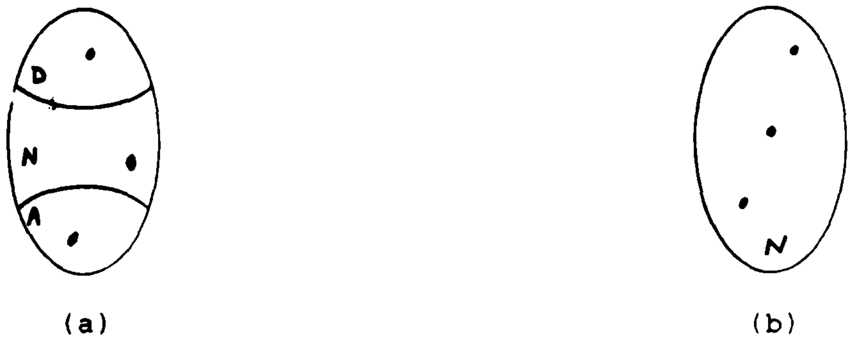


Figure 3. Two Partitions of a Three-valued Logic System.

Example 2-4. Given a designated logic system of four values, the three possible partitions of its logic values are shown in figure 4. Figure 4 (a) is the partition of Lukasiewicz' four valued logic (L4). The reader will find a complete definition of L4 in Appendix 1.

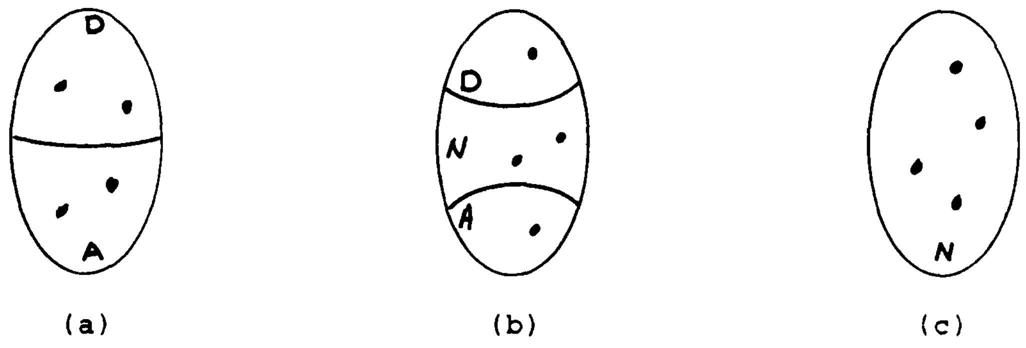


Figure 4. Three Partitions of a Four-valued Logic System.

Because each designated value in a designated logic system must have an associated antidesignated value, there are $\lfloor n/2 \rfloor + 1$ ($\lfloor n/2 \rfloor$ represents the 'floor' function applied to n/2) possible partitions for any set of n elements (n > 1). If n is even the set N can have 0, 2, 4, ... n elements while if n is odd the set N can have 1, 3, 5, ... n elements.

2.7 The CREATE-LOGIC Program.

CREATE-LOGIC is a LISP program written to illustrate some of the principles discussed in this chapter. The source code for CREATE-LOGIC may be found in Appendix 2. Using the program CREATE-LOGIC, we create and store logic values and functions for later use with other LISP programs written for this paper. In the following examples we create files for Boolean logic, L3 and others. The logic functions are stored in file "b:<lname>.lsp" where <lname> is the logic name.

```
> (load 'b:thesis)
; loading "B:THESIS.lsp"
T
> (create-logic)

Enter logic name: > bool                           ;create classical logic

Enter logic function:
> (~ ((T) F)                                       ;negation
1>    ((F) T))

Enter logic function:

> (^ ((T T) T)                                     ;conjunction
1>    ((T F) F)
1>    ((F T) F)
1>    ((F F) F))

Enter logic function:

> (v ((T T) T)                                     ;disjunction
1>    ((T F) T)
1>    ((F T) T)
1>    ((F F) F))

Enter logic function:
```

```
> (=> ((T T) T)                                     ;implication
1>    ((T F) F)
1>    ((F T) T)
1>    ((F F) T))

Enter logic function:

> (<=> ((T T) T)                                    ;equivalence
1>     ((T F) F)
1>     ((F T) F)
1>     ((F F) T))

Enter logic function:

> nil
NIL
>
```

The functions of Boolean logic are now stored in file "*b:bool.lsp*" and can be retrieved as necessary. Similarly, we will create a file to store the functions and logic values of L3.

```
> (create-logic)

Enter logic name: > L3                              ;create Luk's L3

Enter logic function:
> (~ ((T) F)                                        ;negation
1>   ((I) I)
1>   ((F) T))

Enter logic function:

> (^ ((T T) T)                                      ;conjunction
1>   ((T I) I)
1>   ((T F) F)
1>   ((I T) I)
1>   ((I I) I)
1>   ((I F) F)
1>   ((F T) F)
1>   ((F I) F)
1>   ((F F) F))

Enter logic formula:

> (v ((T T) T)                                  ;disjunction
1>   ((T I) T)
1>   ((T F) T)
1>   ((I T) T)
1>   ((I I) I)
1>   ((I F) I)
1>   ((F T) T)
1>   ((F I) I)
1>   ((F F) F))

Enter logic formula:

> (=> ((T T) T)                                 ;implication
1>    ((T I) I)
1>    ((T F) F)
1>    ((I T) I)
1>    ((I I) T)
1>    ((I F) I)
1>    ((F T) T)
1>    ((F I) T)
1>    ((F F) T))

Enter logic formula:

> (<=> ((T T) T)                                ;equivalence
1>     ((T I) I)
1>     ((T F) F)
1>     ((I T) I)
1>     ((I I) T)
1>     ((I F) I)
1>     ((F T) F)
1>     ((F I) I)
1>     ((F F) T))

Enter logic formula:
```

```
> nil
NIL
>
```

2.8 Summary.

Logic systems can be partitioned into Designated, Antidesignated and Neutral classes of truth values. Because the negation function takes designated values into antidesignated values and antidesignated values into designated values, we can use negation to partition the logic values of a logic system. If a logic system has a partition of its logic values meeting these constraints, we call it a Designated Logic System. A Designated Logic System of n logic values must have one of $\lfloor n/2 \rfloor + 1$ possible partitions of its logic values.

CHAPTER 3

ASSIGNING LOGIC CLASSES USING COMPLEMENTATION

As noted in chapter two, we can use the negation function of a logic system to find the partition of a designated logic system. In classical logic we call the negation function 'not.' In logic systems with more than two truth values it may be unclear what we mean by the term 'not x.' For example, given the set {a,b,c}, 'not a' could mean 'either b or c or both,' being a specific exclusion of the element "a" rather than a function mapping designated values to antidesignated values. To avoid possible confusion we will call the negation function 'complement' rather than 'negation' when talking about designated logic systems. Thus, ~x will read 'the complement of x' rather than 'not x.'

If the complement function of a Designated Logic System is a permutation, we can find the system's partition by specifying its cycles. Complement functions which are not permutations are of no interest to this paper.

3.1 Complement Cycle Patterns

A permutation is a function of a set onto itself. In a set of n distinct elements there are n! permutations of those elements. For example, given a set of two elements {T,F}, there are two permutations: $\begin{pmatrix} T & F \\ T & F \end{pmatrix}$ and $\begin{pmatrix} T & F \\ F & T \end{pmatrix}$. Permutation $\begin{pmatrix} T & F \\ T & F \end{pmatrix}$ means T maps to T and F maps to F while permutation $\begin{pmatrix} T & F \\ F & T \end{pmatrix}$ means T maps to F and F maps to T.

Every permutation is uniquely defined by its cycles. Given a permutation function F, a cycle is an ordered list of elements $(x_1\ x_2\ \ldots x_i)$ such that $F(x_1) = x_2$, $F(x_2) = x_3$, ... $F(x_{i-1}) = x_i$ and $F(x_i) = x_1$. In classical logic ~T = F and ~F = T so the only cycle is (T F).

Let $C_{i,j}$ represent the jth cycle of i elements in a permutation. If there are k cycles of length i then we say $N(C_i) = k$. The set of $N(C_i)$'s represented as a sum of individual elements in a permutation is the permutation's cycle pattern. For example, if a permutation of four elements has two cycles of length one and one cycle of length two, its cycle pattern would be 1 + 1 + 2. Although there are n! permutations of n elements, not all the cycle patterns are distinct.

Example 3-1. Given a set of three elements {a,b,c} there are six possible permutations: $\begin{pmatrix} a & b & c \\ a & b & c \end{pmatrix}$, $\begin{pmatrix} a & b & c \\ a & c & b \end{pmatrix}$, $\begin{pmatrix} a & b & c \\ b & a & c \end{pmatrix}$, $\begin{pmatrix} a & b & c \\ b & c & a \end{pmatrix}$, $\begin{pmatrix} a & b & c \\ c & a & b \end{pmatrix}$ and $\begin{pmatrix} a & b & c \\ c & b & a \end{pmatrix}$. As can be seen in figure 5, there are only three unique cycle patterns out of the six possible permutations.

(a)(b)(c)	$C_{1.1} = (a)$	$C_{1.2} = (b)$	$C_{1.3} = (c)$	$N(C_1) = 3$	
(a)(b c)	$C_{1.1} = (a)$	$C_{2.1} = (b\ c)$		$N(C_1) = 1$	$N(C_2) = 1$
(a b)(c)	$C_{1.1} = (c)$	$C_{2.1} = (a\ b)$		$N(C_1) = 1$	$N(C_2) = 1$
(a b c)	$C_{3.1} = (a\ b\ c)$			$N(C_3) = 1$	
(a c b)	$C_{3.1} = (a\ c\ b)$			$N(C_3) = 1$	
(a c)(b)	$C_{1.1} = (b)$	$C_{2.1} = (a\ c)$		$N(C_1) = 1$	$N(C_2) = 1$

Figure 5. Cycle Patterns for a Set of Three Elements.

The number of distinct cycle patterns for a set of n elements is equal to the number of ways n non-distinct objects can be placed into n non-distinct cells with empty cells allowed (this is a restatement of the integer partition enumeration problem). Using a generating function we can calculate the number of distinct cycle patterns for n elements as the coefficient of the term X^n in the following equation [Liu, 1968].

$$\frac{1}{\prod_{i=1}^{n} (1 - X^{i})} \qquad (3\text{-}1)$$

Because the number of cycle patterns is finite for any n elements, we can associate a large number of logics with relatively few cycle patterns.

Example 3-2. In a set of five logic values there are seven cycle patterns. We can find them by enumeration as shown in figure 6.

1+1+1+1+1	$N(C_1) = 5$				
1+1+1+2	$N(C_1) = 3$	$N(C_2) = 1$			
1+2+2	$N(C_1) = 1$	$N(C_2) = 2$			
1+1+3	$N(C_1) = 2$		$N(C_3) = 1$		
2+3		$N(C_2) = 1$	$N(C_3) = 1$		
1+4	$N(C_1) = 1$			$N(C_4) = 1$	
5					$N(C_5) = 1$

Figure 6. Cycle Patterns for a Set of Five Elements.

We can derive the same result using equation 3-1 with n = 5. The coefficient of the X^5 term in equation 3-2 will be the number of cycle patterns possible for a set of five elements.

$$\frac{1}{(1 - X)(1 - X^{2})(1 - X^{3})(1 - X^{4})(1 - X^{5})} \qquad (3\text{-}2)$$

When expanded, equation 3-2 is the polynomial $1 + X + 2X^2 + 3X^3 + 5X^4 + 7X^5 + . . .$ The coefficient of X^5 is seven, so there are seven possible cycle patterns of permutations of five elements.

Although equation 3-1 will allow us to find the number of different cycle patterns, it will not tell us what those patterns are. We can develop a recursive procedure to generate the cycle patterns of n elements given the cycle patterns of n-1 elements. Assume we are given a cycle pattern of n-1 elements in the form a + a + ... + a + b + b + ... + b + ... k = n - 1. From it we can generate k + 1 cycle patterns of n elements:

```
1 + a + a + ... + a       + b + ... + b + ...       + k
    a + a + ... + (a + 1) + b + ... + b + ...       + k
    a + a + ... + a       + b + ... + (b + 1) + ... + k
                                .
                                .
                                .
    a + a + ... + a       + b + ... + b + ...       + (k + 1)
```

Some of the k + 1 cycle patterns will be repeated in several expansions of n - 1 elements. For example, given the five cycle patterns of n = 4 elements we can transform them into the seven possible cycle patterns of n = 5 elements.

1 + 1 + 1 + 1 => 1 + 1 + 1 + 1 + 1
1 + 1 + 1 + 2
1 + 1 + 2 => ~~1 + 1 + 1 + 2~~ already generated
1 + 2 + 2
1 + 1 + 3
1 + 3 => ~~1 + 1 + 3~~ already generated

		2 + 3	
		1 + 4	
2 + 2	=>	~~1 + 2 + 2~~	already generated
		~~2 + 3~~	already generated
4	=>	~~1 + 4~~	already generated
		5	

Figures 7 (a) through 7 (g) show the seven cycle patterns for five elements. In the next section we will discuss how cycle patterns determine the partitions shown in figure 7.

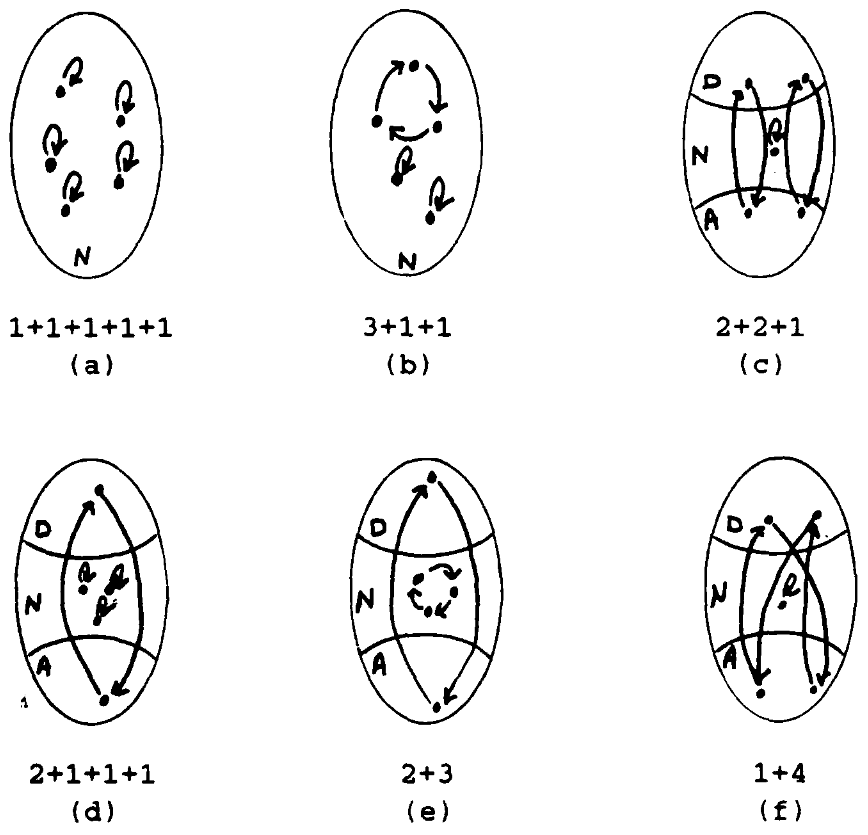
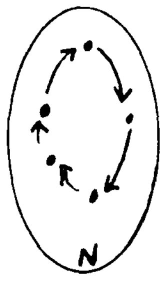


Figure 7. Cycle Patterns for a Set of Five Elements.



5
(g)

Figure 7 - Continued. Cycle Patterns for a Set of Five Elements.

3.2 Partitioning Constraints Using Cycles.

Using constraints imposed on elements in classes D, A and N in chapter two, we can see that all elements in odd cycles (cycles containing an odd number of elements) must be in the set N: given the cycle (x_1 x_2...x_{2i+1}), if x_1 is in D then x_2 is in A and x_3 is in D and x_5 is in D and ... and x_{2i+1} is in D and x_1 is in A. This contradicts the first assumption since D and A are disjoint. Likewise, we could prove that if x_1 is in A then x_1 is in D for x_1 in an odd cycle. Therefore all elements in odd cycles must be in the set N.

Although there is no pressing mathematical necessity to place elements of even cycles in classes D and A, we will do so whenever possible to simplify the classification and partitioning process. If we do not require even cycles to be placed in D and A whenever possible, we increase the number of possible assignments for a complement function with k even cycles from 2^K to

$$\sum_{i=1}^{K} \binom{K}{i} 2^{K-i}.$$

There are cases, as we will see in Chapter 4, when it is not possible to put all even cycles in sets D and A. These are special cases and do not increase the numerical complexity of the classification procedure.

Given an even cycle $(x_1\ x_2\ \ldots\ x_{2i})$, if $\{x_{2j+1} \mid 0 \leq j \leq i-1\}$ is a subset of D then $\{x_{2j} \mid 1 \leq j \leq i\}$ is a subset of A. Likewise, if $\{x_{2j+1} \mid 0 \leq j \leq i-1\}$ is a subset of A then $\{x_{2j} \mid 1 \leq j \leq i\}$ is a subset of D.

Example 3-3. Given a Post logic of three elements (P3), we can find its cycle pattern and its partition using the definition of complementation in table 2.

.able 2

Complementation in P3

	$\bar{}$
1	2
2	3
3	1

This permutation has only one cycle of three elements, so its cycle pattern is 3. Also, since there are only odd cycles, all logic values in P3 must be in N as shown in figure 8. In general, any odd numbered Post system will have cycle pattern n with all elements in class N.

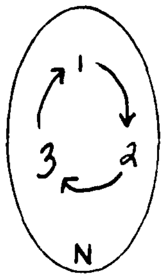


Figure 8. P3 With Cycle Pattern 3.

Example 3-4. Given complementation in Kleene's three-valued logic (K3) as shown in table 3, we can determine its cycle pattern and its partition but we are unable to assign specific logic values to each partition class.

Table 3

Complementation in K3

~	
T	F
I	I
F	T

There are two cycles in K3's complement permutation. (I) is a cycle of one element and (T F) is a cycle of two elements, so the cycle pattern of K3 is 2+1. Although we know K3's cycle pattern and partition, we are unable to determine a unique assignment of K3's elements to specific classes within the partition. Until we attach further meaning to logic values in Designated Logic Systems, all possible assignments of elements in even cycles to partition classes are isomorphic. Both possible assignments of K3's elements are shown in figure 9.

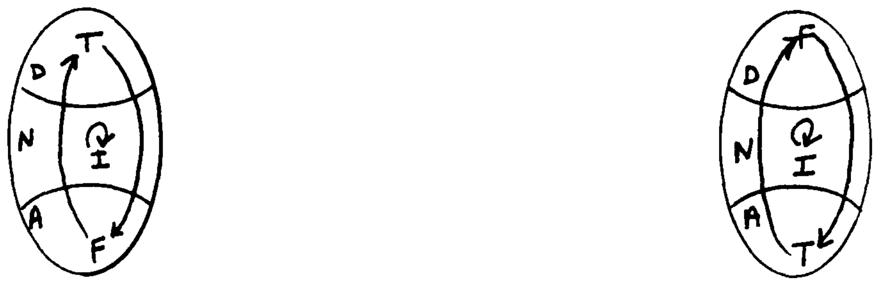


Figure 9. Two Possible Partitions of K3.

3.3 Cycle-classes of Logic Elements.

Given a complement permutation with an even cycle (a_1 a_2

are in the same class, as are elements a_2, a_4 ... a_{2i}. We define the term cycle-class to be the set of elements in both the same cycle and the same class. We can use the cycle-class specification of logic values to preserve the complementary nature of elements in even cycles without the necessity of determining the specific class of individual elements. This allows us to circumvent the problem of discussing logic values in terms of their isomorphic representations similar to those in example 3-4.

Example 3-5. Given Moussavi's six-valued logic (M6), detailed in appendix 1, with the complement permutation defined by table 4, there are six cycle-classes of the six elements since each element is in its own cycle-class.

Table 4

Complementation in M6

~	
T	F
K0	K1
U	U
K	K
K1	K0
F	T

Example 3-6. Given the complement permutation in a Post logic of four elements as defined in table 5, there are two cycle-classes: {1,3} and {2,4}.

Table 5

Complementation in P4

~	
1	2
2	3
3	4
4	1

In general, all elements in an odd cycle are in the same cycle-class while elements in even cycles form two complementary cycle-classes. If a complement permutation has j odd cycles and k even cycles, the logic system has j + 2k cycle-classes in all.

3.4 The FIND-CYCLES Program.

The LISP program FIND-CYCLES finds the odd and even cycles in a logic using the complement function. The logic must have been created using the CREATE-LOGIC program described in chapter two, and the complement function must be defined by the symbol ~.

If the complement function is a permutation, FIND-CYCLES returns a list of the even and odd cycles. If the complement function is not a permutation, a message to that effect is sent to the user. The LISP code for FIND-CYCLES is in appendix 3.

Below are some examples using FIND-CYCLES on various logics we defined using CREATE-LOGIC. In the first example we apply FIND-CYCLES to Classical logic and the results are as expected: no odd cycles and one even cycle.

```
> (find-cycles)

Enter logic name: > bool
; loading "b:BOOL.lsp"
(NIL ((T F)))
> odd
NIL
> even
((T F))
```

When we apply *FIND-CYCLES* to an odd numbered Post system we get opposite results: no even cycles and one odd cycle.

```
> (find-cycles)

Enter logic name: > Post3
; loading "b:POST3.lsp"
(((1 2 3)) NIL)
> odd
((1 2 3))
> even
NIL
```

When we apply FIND-CYCLES to a logic with multiple odd and even cycles such as M6, neither the set of even cycles nor the set of odd cycles is NIL.

```
> (find-cycles)

Enter logic name: > sixval
; loading "b:SIXVAL.lsp"
(((U) (K)) ((T F) (K1 K0)))
> odd
((U) (K))
```

```
> even
((T F) (K1 K0))
```

When we apply FIND-CYCLES to a complement function that is not a permutation, we receive an error message. The logic non-perm has a complement function as shown in table 6. Since the function is not a permutation, FIND-CYCLES returns an error message.

Table 6

A Complement Function Which is Not a Permutation

~	
T	F
U	F
F	T

```
> (find-cycles)

Enter logic name: > non-perm
; loading "b:NON-PERM.lsp"
not a permutation
NIL
> odd
NIL
> even
NIL
```

We will use the sets of odd and even cycles in the LISP program FIND-ASSIGNMENT explained in chapter 4 to determine the assignment of logic values to classes D and A. The elements in odd cycles are already known to be in class N.

3.5 Summary.

Given the complement permutation of a logic, we can associate with it a specific cycle pattern. The cycle pattern of the complement function is sufficient to specify a partition of the designated logic, but not always sufficient to determine a unique assignment of logic values to partition classes. We know elements of even cycles will be in either the set D or the set A, while all elements of odd cycles will be in set N, so any logic with only odd cycles will have all elements in N. Elements of even cycles will have more than one possible assignment as we saw in example 3-4. In a logic system with k even cycles there are 2^k isomorphic assignments of elements to classes D and A.

Because we cannot specify a unique assignment of logic elements in even cycles we use cycle-classes of the logic elements. The use of cycle-classes when referring to elements in even cycles allows us to preserve the complementary nature of elements in even cycles without regarding problems of unique identification posed by the isomorphic assignments possible using only complementation as an assignment constraint.

The program FIND-CYCLES finds the odd and even cycles in a complement permutation. We will use this information later to discover assignments of logic values to partition classes.

In the next chapter we will investigate an assignment scheme based on the conjunction function together with methods defined in this chapter and develop an algorithm for determining the assignment of elements in Designated Logic Systems to the partition classes D, N and A.

CHAPTER 4

FINDING A PARTITION ASSIGNMENT USING CONJUNCTION

In the previous chapter we discovered that complementation did not suffice to assign all logic values to a unique partition class in a Designated Logic System. In this chapter we will use the complement permutation together with the conjunction function to uniquely determine the assignment of values in a designated logic system to within equivalence.

4.1 Conjunction in Designated Logic Systems.

In classical logic, the conjunction of two elements is not true if at least one argument is false. We would like to generalize the definition of conjunction in Designated Logic Systems. Given a designated logic system with conjunction function &, if x & y is designated (i.e. an element of D) then both x and y are designated (i.e. x, y are elements of D). Otherwise the conjunction function is unconstrained. Any conjunction function meeting these constraints is a well-defined conjunction function. If we are given a conjunction function which does not meet these constraints we say it is an ill-defined conjunction function.

In a system with k even cycles in the complement permutation there are 2^k possible assignments of elements to classes of the partition since every even cycle-class may be either in D or in A. As a result of our definitions, any assignment of logic elements for which conjunction is well defined is a possible assignment of the logic elements within a Designated Logic System.

In chapter three we were unable to determine a unique assignment for Kleene's three-valued logic using only the cycle pattern imposed by complementation. Using the definition of conjunction in K# we can determine the only possible assignment of logic values to classes of the partition.

Example 4-1. Given Kleene's three-valued logic with the conjunction and complement functions defined in table 7, either T is designated and F antidesignated, or T is antidesignated and F designated (see example 3-4). I, of course, is neutral.

Table 7

Complementation and Conjunction in K3

~	
T	F
I	I
F	T

&	T	I	F
T	T	I	F
I	I	I	F
F	F	F	F

If we assign T to the designated class and F to the antidesignated class, the conjunction function shown in table 7 is well defined since a & b is designated only when a = b = T. If we assume the alternate assignment (F designated and T antidesignated), the conjunction function is not well defined since T and F are not both designated yet T & F = F. Because T designated and F antidesignated is the only assignment in K3 for which conjunction is well defined, the only possible assignment of K3's logic values can be seen in figure 10.

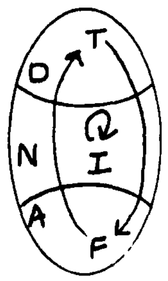


Figure 10. The Partition of K3.

Example 4-2. Given Bochvar's three-valued logic (system B3) with complementation and conjunction as shown in table 8, we see that either T is designated and F antidesignated, or F is designated and T antidesignated. Again, I is neutral.

Table 8

Complementation and Conjunction in B3

~	
T	F
I	I
F	T

&	T	I	F
T	T	I	F
I	I	I	I
F	F	I	F

If we assume T is designated and F antidesignated, the conjunction function defined in table 8 is well defined. Alternately, if F is designated and T antidesignated, conjunction is not well defined (T & F = F yet T and F are not both designated). The only possible assignment of B3's logic values in a Designated Logic System, as shown in figure 11, is D = {T}, A = {F} and N = {I}.

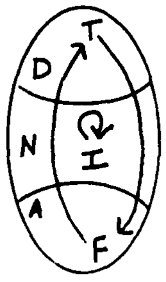


Figure 11. The Partition of B3.

Example 4-3. If we have Post Logic System P4 with complementation and conjunction defined by table 9, there are two possible assignments of elements to the designated and antidesignated classes. Either D = {1,3} and A = {2,4} or D = {2,4} and A = {1,3} (N is empty since there are no odd cycles in P4).

Table 9

Complementation and Conjunction in P4

~	
1	2
2	3
3	4
4	1

&	1	2	3	4
1	3	3	3	2
2	3	4	4	2
3	3	4	1	2
4	2	2	2	2

An examination of the conjunction table shows that 3 & 4 =

2. Since 3 and 4 are in complementary classes, 2 must be antidesignated. Also, since 2 and 4 must be in the same class (~(~2) = 4), 4 is also antidesignated and A = {2,4}. Since 1 & 2 = 3, and 2 is antidesignated, 3 must be antidesignated. This is a contradiction and we are forced to conclude there are no assignments of elements in P4 for which conjunction is well defined.

We can generalize this conclusion and claim that there is no possible assignment of logic elements in a Post system with an even number of elements greater than two for which conjunction is well defined within the Designated Logic System framework. First, we know it is true of P4 by example 4-3. Assume we have an even numbered Post system Pn where n $\geq$ 6. Because of the nature of complementation in Post systems (<u>i.e.</u> it is cyclic mod n), we know that elements 1 and 3 are in the same class (either D or A) but in a different class than elements 2 and 4. If 1 and 3 are designated then 2 and 4 are antidesignated. Likewise, if 1 and 3 are antidesignated then 2 and 4 are designated. We also know that 1 & 2 = 3, so 1 and 3 must not be designated. Because 2 & 3 = 4 we know 2 and 4 must not be designated, and the system does not have an assignment of logic values for which conjunction is well defined.

There are some logic systems which have more than one possible assignment of logic values preserving the definition of conjunction in Designated Logic Systems. In this situation we are unable to specify a unique assignment of the logic elements.

Example 4-4. Assume we have a designated logic system whose conjunction and complement functions are defined by table 10. We will call this system the Skeptic's three-valued logic system or SL3. We know from the complement permutation that N = {I}. We also know that either T is designated and F antidesignated, or F is designated and T antidesignated.

Table 10

Complementation and Conjunction in SL3

¯	
T	F
I	I
F	T

&	T	I	F
T	T	I	I
I	I	I	I
F	I	I	F

We can not determine a unique assignment of SL3's logic values since both possible assignments preserve the definition of conjunction in Designated Logic Systems. If T is designated, a & b = T if and only if a = b = T. If F is designated, a & b = F if and only if a = b = F. Using conjunction as the determining factor both assignments are possible within the framework of Designated Logic Systems. We will examine these types of systems (i.e. systems with multiple possible assignments) in greater detail in section 4.2.

Although we can now uniquely determine the assignment of many more Designated Logic Systems, there are some systems for which we need additional information. Logic systems with ill-defined conjunction (conjunction is not well defined for any assignment of logic values to partition classes) and logic systems with multiple possible assignments are two such examples.

4.2 Equivalent Partitions.

As we saw in example 4-4, some logic systems have more than one possible assignment yielding well defined conjunction functions. These logic systems have equivalent partitions.

Given a cycle whose logic values we can not uniquely assign to partition classes, we know the cycle is even since all odd cycles are in N. As we saw in chapter two, every even cycle is divided into two complementary cycle-classes. If we can map elements of an even cycle to elements in its complementary cycle-class such that the conjunction function is identical, the two possible partitions are equivalent.

Assume we have a logic system whose complement function has an even cycle $c_{i,j}$. If we can find a function f: S -> S which maps elements of $c_{i,j}$ to elements in its complementary cycle-class while all other elements in the logic are mapped to themselves such that f(x) & f(y) = f(x & y) for all x and y in the logic, then the two partitions are equivalent.

Example 4-5. Given the SL3 logic defined in example 4-4 with the mapping f(T) = F, f(F) = T and f(I) = I, the assignment D = {T}, A = {F}, N = {I} is equivalent to the assignment D = {F}, A = {T}, N = {I} since the two conjunction functions are the same (see table 11).

Table 11

Equivalent Conjunction Functions in SL3

&	T	I	F
T	T	I	I
I	I	I	I
F	I	I	F

&	F	I	T
F	F	I	I
I	I	I	I
T	I	I	T

It is possible to have a logic where some even cycles are specifically assigned while other even cycles are not.

Lukasiewicz' four-valued logic (L4) is such a system.

Example 4-6. Logic system L4 has conjunction and complementation as defined in table 12. Using the complement permutation we know TT and FF are in complementary classes as are FT and TF. Because FF & TT = FF, we know FF is antidesignated and TT is designated. Conjunction is well defined both when FT is designated and TF antidesignated, and when TF is designated and FT antidesignated. Therefore there are two possible assignments of the elements in L4 to partition classes.

Table 12

Complementation and Conjunction in L4

~	
TT	FF
TF	FT
FT	TF
FF	TT

&	TT	TF	FT	FF
TT	TT	TF	FT	FF
TF	TF	TF	FF	FF
FT	FT	FF	FT	FF
FF	FF	FF	FF	FF

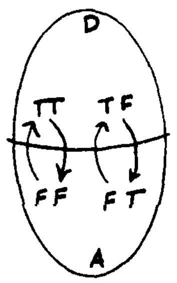
Let $\hat{f}$ be a mapping such that f(TT) = TT, f(FF) = FF, f(TF) = FT and f(FT) = TF. The two conjunction functions are equal as shown in table 13. Therefore, the two possible assignments shown in figure 12 are equivalent.

Table 13

Equivalent Conjunction Functions in L4

&	TT	TF	FT	FF
TT	TT	TF	FT	FF
TF	TF	TF	FF	FF
FT	FT	FF	FT	FF
FF	FF	FF	FF	FF

&	TT	FT	TF	FF
TT	TT	FT	TF	FF
FT	FT	FT	FF	FF
TF	TF	FF	TF	FF
FF	FF	FF	FF	FF



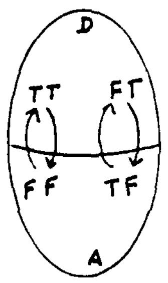


Figure 12. Equivalent Assignments in L4.

Example 4-7. There are no equivalent assignments in Kleene's three-valued logic. The only even cycle is (T F), so the only possible mapping is f(T) = F, f(I) = I and f(F) = T. As we can see from table 14, the two conjunction functions are not equal, so the two assignments are not equivalent.

Table 14

Non-Equivalent Conjunction Functions in K3

&	T	I	F
T	T	I	F
I	I	I	F
F	F	F	F

&	F	I	T
F	F	I	T
I	I	I	T
T	T	T	T

Example 4-8. There are four possible mappings of elements to their complementary cycle-class in a Post system of four elements with conjunction and complementation as defined in example 4-3. The four mappings are listed in table 15.

Table 15

Four Mappings of Complementary Cycle-Classes in P4

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
1	2	4	2	4
2	1	3	3	1
3	4	2	4	2
4	3	1	1	3

Table 16 shows the resulting conjunction functions; none are equal to conjunction in P4. Hence, there are no equivalent assignments in P4.

Table 16

Non-Equivalent Functions in P4

&	1	2	3	4
1	3	3	3	2
2	3	4	4	2
3	3	4	1	2
4	2	2	2	2

(a)
Conjunction in P4

&	2	1	4	3
2	4	4	4	1
1	4	3	3	1
4	4	3	2	1
3	1	1	1	1

(b)
$f_1(x)$

&	4	3	2	1
4	2	2	2	3
3	2	1	1	3
2	2	1	4	3
1	3	3	3	3

(c)
$f_2(x)$

&	2	3	4	1
2	4	4	4	3
3	4	1	1	3
4	4	1	2	3
1	3	3	3	3

(d)
$f_3(x)$

&	4	1	2	3
4	2	2	2	1
1	2	3	3	1
2	1	3	4	1
3	1	1	1	1

(e)
$f_4(x)$

Using equivalent conjunction functions we can classify some logic systems whose values we couldn't uniquely assign. All Designated Logic Systems with well defined conjunction functions have a unique assignment up to equivalence.

Logics with no assignment preserving the definition of conjunction in Designated Logic Systems continue to elude us. In the next section we will learn more about these systems in an attempt to find a method of imposing an assignment on their logic values.

4.3 Identity Elements and the Supremum.

An identity element in a mathematical system with a binary operator * is defined as an element s such that s * x = x * s = x for all elements x within the system. In a designated logic system we will call element s an identity element if s & x = x & s = x' for all x in the logic where x' is in the same cycle-class as x. If s is unique we will call s the supremum of the logic system.

Every identity element must be in the same cycle class. Assume we have a logic system with two identity elements: s1 and s2. Also assume s1 and s2 are not in the same cycle-class. By the definition of identity elements s1 & x = x & s1 = x' where x and x' are in the same cycle-class therefore s1 & s2 = s2 & s1 = s2' where s2 and s2' are in the same cycle-class. By the definition of identity elements s2 & x = x & s2 = x' where x and x' are in the same cycle-class. Therefore s2 & s1 = s1 & s2 = s1' where s1 and s1' are in the same cycle-class. We must conclude that s1' = s2' and s1 and s2 are in the same cycle-class. This contradicts our assumption that s1 and s2 were in different cycle-classes, so all identity elements must be in the same cycle-class.

Example 4-9. P4 has two cycle-classes, {1,3} and {2,4}.

From table 17 we see 3 & x = x' for all x in S. Therefore element 3 is an identity element. Also, 3 is the only element in P4 with that property, so 3 is also the supremum.

Table 17

The Supremum in P4

&	1	2	3	4
1	3	3	3	2
2	3	4	4	2
3	3	4	1	2
4	2	2	2	2

In any even numbered Post system Pn, the supremum will be n-1. By the definition of conjunction in Post systems, we know that x & (n-1) = ~(min(~x,~(n-1))) = ~(min((x+1) mod n,n). Since all elements of Pn must be less than or equal to n, min((x+1) mod n,n) = (x+1) mod n = ~x. Therefore, x & (n-1) = ~~x which is in the same cycle-class as x, so n-1 is the supremum in any even Post system.[4]

If a logic system has a conjunction function preserving the definition of conjunction in Designated Logic Systems, and at least one even cycle, all identity elements (including the supremum if it exists) will be designated. Because we assume the logic system has an even cycle we know that set D is not empty and set A is not empty. Assume s is an identity element but s is not designated. Also assume element x is designated (we know x exists because D is not empty). Because x & s = x' where x and x' are in the same cycle-class, the result x & s must be designated. This contradicts the definition of conjunction: the result can be designated only if all of its arguments are designated. Therefore s must be designated.

If an identity element is in a cycle of length two, it is the strict identity element we call the supremum. We know that all identity elements must be in the same cycle-class.

Example 4-10. In L4 TT & x = x & TT = x for all x in the logic system. TT is not only an identity element, it is the supremum since no other element has that property (see table 18).

Table 18

The Supremum in L4

&	TT	TF	FT	FF
TT	TT	TF	FT	FF
TF	TF	TF	FF	FF
FT	FT	FF	FT	FF
FF	FF	FF	FF	FF

Although we can identify the supremum element in a Designated Logic System if the supremum exists, we cannot always determine its class. If conjunction is well defined and there is at least one even cycle in the complement function, the supremum will be designated. If all logic values are in N, any identity elements will be in N. In a system whose conjunction function is not well defined, the supremum could be in any class (e.g. the even Post systems of more than two elements). We must conclude that identifying the supremum will not enable us to specify an assignment of the logic's elements.

4.4 Zero Elements and the Infimum.

In mathematical systems with a binary operation *, a zero element z has the property z * x = x * z = z for all x in the system. We will define a zero element in Designated Logic Systems as an element z such that x & z = z & x = z' for all x in the logic where z and z' are in the same cycle-class. If the zero element is unique, we will call it the infimum of the logic system.

All zero elements must be in the same cycle-class. Assume we have two zero elements z1 and z2 not in the same cycle-class. By our definition of a zero element z1 & x = x & z1 = z1' where z1 and z1' are in the same cycle-class. Therefore z1 & z2 = z2 & z1 = z1'. By the same definition of a zero element z2 & x = x & z2 = z2' where z2 and z2' are in the same cycle-class. Therefore z2 & z1 = z1 & z2 = z2'. Because z1' = z2' z1 and z2 must be in the same cycle-class which contradicts our assumption so all zero elements must be in the same cycle-class.

Example 4-11. In P4 with conjunction and complementation as defined in example 4-3, 4 is a zero element. As shown in table 19, 4 & x = 2 for all x in P4, and 2 and 4 are in the same cycle-class. Since 4 is the only element with this property, it is also the infimum.

Table 19

The Infimum in P4

&	1	2	3	4
1	3	3	3	2
2	3	4	4	2
3	3	4	1	2
4	2	2	2	2

We can generalize the above result and say n is the infimum for any Post system of n elements where n is even. In a Post system, n & x = ˜(min(˜n,˜x)) = ˜(min(1,(x+1) mod n)) = ˜1 =

2. Since n and 2 are in the same cycle-class (2 = ~~n), element n is the infimum.

When a zero element is in a cycle of length two, it is a strict zero element where z & x = x & z = z for all x in he logic system. If z is in a cycle of length two, z is in a cycle-class containing itself alone. Therefore if z & x = z' where z and z' are in the same cycle-class, z must be equal to z' and z & x = z.

Example 4-12. In L4 element FF is in a cycle of length two. Also, FF & x = x & FF = FF for all x in the logic (see table 20) so FF is a strict zero element.

Table 20

The Infimum in L4

&	TT	TF	FT	FF
TT	TT	TF	FT	FF
TF	TF	TF	FF	FF
FT	FT	FF	FT	FF
FF	FF	FF	FF	FF

Although we can identify the infimum if such an element exists, we cannot determine its class other than to note it can not be designated when conjunction is well defined. Assume z is designated and some element x is not designated (either in N or A). By our definition of a zero element, z & x = z' must be designated. This contradicts our definition of well defined conjunction, so z must not be in D.

When we have a system with well defined conjunction, we can determine the class of the infimum. If it is in an even cycle, the infimum is antidesignated since it can not be designated. If it is in an odd cycle, the infimum is neutral since all elements of odd cycles are in N. If we have a system with ill-defined conjunction, identifying the infimum does not help us find an assignment for the logic elements since z could be in any of the three partition classes.

4.5 An Algorithm for Partitioning All Designated Logic Systems.

Thus far we are able to determine a unique assignment of a system's logic values up to equivalence when the conjunction function is well defined. There are, however, some logic systems we were unable to assign. Logic systems with no possible assignment of logic values to partition classes for which conjunction is well defined cannot be partitioned using constraints we have discussed.

There are several possible solutions to this problem. We can ignore logics having no possible assignments, eliminating them from our discussion. We could define the class in which the supremum resides as designated, or we could specify different classes for logics with ill-defined conjunction functions.

The first solution, eliminating logics with no possible assignments from our discussion, is very tempting. Clearly, if the system does not have conjunction within our constraints, that system will have little in common with classical logic. The principle of contradiction would not hold in these logics since there could be a value in the logic, call it X, such that ˜(X & ˜X) is antidesignated. I would use this pruning method only as a last resort since we do not want to discount the Post systems from our discussion.

The second solution requires that there be a supremum in the logic system, something we can't guarantee. This solution would allow us to admit the Post systems since they do have a supremum, however, we would no longer have any 'principle of conjunction' (if a & b is designated then both a is designated and b is designated) for Designated Logic Systems (see example 4-3). Again, we would consider this solution only if there were no other viable alternatives.

In the third solution we would redefine the classes for systems with ill-defined conjunction. For example, rather than classes D, N and A we could define the supremum class, the infimum class and the neutral class. The logic systems with well defined conjunction would be a subset of these logics where the supremum class is designated, etc. This classification would put the Post systems in a 'separate but equal' category from the designated logic systems. This is better than eliminating them because it allows us to admit the fundamental difference between Post systems and other logics more closely related to classical logic. This solution seems to solve our difficulties with the Post systems. In truth, however, it is merely a covert implementation of the first solution, i.e. to eliminate troublesome logics from our discussion.

Finally, we could elect to put all logic values contributing to ill-defined conjunction into the neutral class, along with other elements in its cycle. For example, in a Post system of more than two values, all elements would be in N. This solution will allow any possible system to be classified as a Designated Logic System since, in the wors case, conjunction is well defined when all logic values are in the neutral class. We lose the ability of determining the logic's partition using the cycles of the complement function, however, yet this seems a lesser evil than arbitrary pruning.

We can now develop an algorithm for partitioning a Designated Logic System. First, use the cycles of the complement function to determine the set of possible partitions. If all the cycles are odd, the only partition possible is one in which all elements are in the neutral class. Given k even cycles there are 2^k possible assignments of the logic values, assuming conjunction is well defined for some assignment.

Next, using the conjunction function, determine all possible assignments preserving the definition of conjunction in designated logic systems. If there is more than one assignment meeting this constraint, they are equivalent. If there is no possible assignment for which conjunction is well defined, place the even cycles causing ill-defined behavior in the neutral class. In the worst case all values will be placed in the neutral class.

Example 4-13. Assume we have a logic system with conjunction and complementation as defined in table 21. We will call this logic system ILL6.

Table 21

Complementation and Conjunction in ILL6

~	
T	F
T1	F2
T2	F1
F1	T1
F2	T2
F	T

&	T	T1	T2	F1	F2	F
T	T	T1	T2	F1	F2	F
T1	T1	T1	T2	F1	F2	F
T2	T2	T1	T2	T2	F2	F
F1	F1	F1	T2	F1	F2	F
F2	F2	F2	F2	F2	F2	F
F	F	F	F	F	F	F

The conjunction function shows two even cycles, one of length two and one of length four. Elements T and F can be easily partitioned into T in the designated set and F in the antidesignated set since T & F = F but a & b = T if and only if a = b = T.

Elements T1, T2, F1 and F2 can be classified into either T1 and T2 designated with F1 and F2 antidesignated, or T1 and T2 antidesignated with F1 and F2 designated. Since F1 & T1 = F1, F1 and F2 must be antidesignated with T1 and T2 designated. We note, however, that F1 & T2 = T2, so T1 and T2 can't be designated. Since there is no assignment of elements T1, T2, F1 and F2 into classes D and A for which we have well defined conjunction, we must reassign its even cycles. If we put cycle (T F) in the neutral class, there will be no assignment of logic values resulting in well defined conjunction. If we place cycle (T1 F2 T2 F1) in N, however, we have well defined conjunction when T is designated and F antidesignated. The resulting assignment is shown in figure 13.

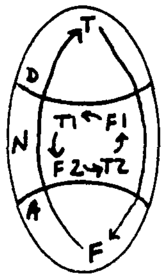


Figure 13. The Partition of ILL6.

4.6 The FIND-ASSIGN Program.

The LISP program FIND-ASSIGN will determine the partition and assignment of any logic system when given a complement permutation and conjunction function. The source code for FIND-ASSIGN may be found in appendix 4.

The following examples of FIND-ASSIGN execution show how the program can determine the partition and assignment of Kleene's logic (as we saw in example 4-1) and Bochvar's logic (as we saw in example 4-2).

```
> (find-assign)

Enter logic name: > kleene
; loading "b:KLEENE.lsp"

Designated values: (T)
Neutral values: (I)
Antidesignated values: (F)
NIL
>
> (find-assign)

Enter logic name: > bochvar
; loading "b:BOCHVAR.lsp"

Designated values: (T)
Neutral values: (I)
Antidesignated values: (F)
NIL
>
```

When there are equivalent partitions, FIND-ASSIGN will choose one semi-arbitrarily. If there are elements 'true,' 'false,' 'T' or 'F' in the logic, FIND-ASSIGN will attempt to select the assignment placing 'true' or 'T' in the designated class and 'false' of 'f' in the antidesignated class (there is no strict mathematical necessity for this assignment, merely English language conventions for the name of logic values). FIND-ASSIGN chooses the assignment of logic values in SL3 (see example 4-4) such that T is designated and F antidesignated.

```
> (find-assign)

Enter logic name: > sl3
```

```
; loading "b:SL3.lsp"

Designated values: (T)
Neutral values: (I)
Antidesignated values: (F)
NIL
>
```

When there is no possible assignment resulting in well defined conjunction, FIND-ASSIGN will selectively place even cycles in the neutral class until an assignment is found. FIND-ASSIGN will place the fewest even cycles in N as possible. In the following example (see example 4-13), FIND-ASSIGN places cycle (T1 F2 T2 F1) in the neutral class, but not cycle (T F).

```
> (find-assign)

Enter logic name: > no-assig
; loading "b:NO-ASSIG.lsp"

Designated values: (T)
Neutral values: (T1 F1 T2 F2)
Antidesignated values: (F)
```

As a last resort, FIND-ASSIGN will place all even cycles in the neutral class. In example 4-3 we saw the Post system of four elements did not have an assignment of its logic values for which conjunction is well defined. FIND-ASSIGN will places all of P4's elements in the neutral class.

```
> (find-assign)

Enter logic name: > p4
```

```
; loading "b:P4.lsp"

Designated values: NIL
Neutral values: (3 2 1 4)
Antidesignated values: NIL
```

4.7 Summary.

In this chapter we have developed a system to find an assignment of logic values to partition classes we defined in chapter two: designated, antidesignated and neutral. This assignment is unique for each logic system up to equivalence within the framework of Designated Logic Systems. The LISP program FIND-ASSIGN will find the assignment of any system given complementation and conjunction.

In the next chapter we will use what we have discovered about these systems to build a hierarchy of mathematical systems characterizing different properties found in Designated Logic Systems.

CHAPTER 5

A HIERARCHY OF DESIGNATED LOGIC SYSTEMS

Although we can now classify designated logic systems based on their partitions and class assignments, we can classify them further based on shared mathematical properties. In this chapter we will explore those shared properties and develop a hierarchy of mathematical systems to classify the designated logic systems. Using these classifications we can find similarities between the Designated Logic Systems and classical logic.

5.1 Designated Algebras.

An algebra is a set of elements S together with a number of operations defined on that set. We call [S,~,&] a designated algebra if S is a set of logic elements, ~ is a complement permutation, as defined in chapter three and & is a well defined conjunction function as defined in chapter four.

We saw from the conclusions in chapter four that every Designated Logic System is also a designated algebra since every set of elements has at least one assignment where conjunction is well defined. Using this property we can show the Principle of Contradiction is a tautology in any Designated Logic System.

Assume there is some value x in a designated logic system such that the principle of contradiction does not hold. In other words, there is some x such that $\tilde{}(x \ \& \ \tilde{}x)$ is antidesignated. This means there is some x such that $x \ \& \ \tilde{}x$ is designated. By the definition of conjunction in Designated Logic Systems we know there is no such x, so the principle of contradiction is a tautology in any Designated Logic System.

5.2 Designated Monoids.

A monoid is a set S together with an associative function and an identity element s such that s & x = x & s = x for all x in the set S. We call [S,˜,&] a designated monoid if it is a designated algebra, contains at least one identity element as defined in chapter three, and & is associative. Any designated logic without an identity would not be a designated monoid. Nor would any system with non-associative conjunction.

Example 5-1. Assume we have a logic system whose conjunction and complement functions are defined by table 22. We will call this logic cyc4 and note that cyc4 has cycle-classes {1 3} and {2 4}. Cyc4 is a designated monoid with identities 1 and 3 (recall the definition of identities in a designated monoid: the identity element need only map elements into the same cycle-class).

Table 22

Complementation and Conjunction in Cyc4

$\bar{}$	
1	2
2	3
3	4
4	1

&	1	2	3	4
1	1	2	3	4
2	2	3	4	1
3	3	4	1	2
4	4	1	2	3

Example 5-2. A Post system of four elements is not a designated monoid. As shown in example 4-9, P4 has the identity element 3. However, since conjunction is not associative in Post systems of more than two elements, P4 is not a designated monoid.

5.3 Ordered Designated Systems.

Let $\leq$ be an order on S such that if x & y = w then w $\leq$ x and w $\leq$ y. This order can be represented as directed graph. If x $\leq$ y then there is a directed path from x to y and x & y = x.

Example 5-3. Let [S,$\bar{}$,&] be a designated monoid with the conjunction function described by table 23.

Table 23

An Ordered Conjunction Function

&	s	y	v	w	x	z
s	s	y	v	w	x	z
y	y	y	x	w	x	z
v	v	x	v	z	x	z
w	w	w	z	w	z	z
x	x	x	x	z	x	z
z	z	z	z	z	z	z

This designated monoid may be represented by the directed graph in figure 14. For example, since y & w = w there is a directed path from y to w and w $\leq$ y.

$$\begin{tikzcd}
& s \arrow[dl] \arrow[dr] & \\
y \arrow[d] \arrow[drr] & & v \arrow[d] \\
w \arrow[dr] & & x \arrow[dl] \\
& z &
\end{tikzcd}$$

Figure 14. A Graph of Table 23.

Given a designated monoid represented as a graph, two elements x and y are comparable (either x $\leq$ y or y $\leq$ x) if and only if there is a directed path from x to y or there is a directed path from y to x. Two elements are incomparable if there is no directed path from x to y and no directed path from y to x.

If, for any two comparable values x and y, $x \leq y$ implies $x \& y = x$, then & is a greatest lower bound function and $\leq$ defines a partial order on the set of logic values. If $\leq$ defines a partial order of the logic elements in a designated monoid, we say [S,˜,&] is an ordered designated system.

Example 5-4. Let S be a set of six elements represented by the integers zero through five. Also, let [S,˜,&] be a designated monoid with the conjunction function defined in table 24.

Table 24

Conjunction For Example 5-3

&	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	4	4	4	5
2	2	4	2	4	4	5
3	3	4	4	4	4	5
4	4	4	4	4	4	5
5	5	5	5	5	5	5

Then $\leq$ defines a partial order on the set S and [S,˜,&] is an ordered designated system represented by the graph in figure 15.

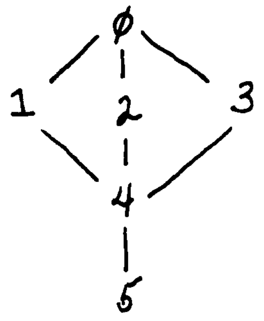


Figure 15. A Graph of Table 24.

We can define another function called least upper bound (lub) as follows. If glb(w,x) = y then lub(w,y) = w and lub(x,y) = x. In other words, given an ordered designated system represented as a directed graph, if there is a directed path from x to y then lub(x,y) = x.

A partially ordered set (poset) is called a lattice if a unique glb and lub exist for every possible pair of elements of S. A Designated Logic System can be represented as a lattice if it is an ordered designated system and the lub function defined disjunction in the Designated Logic System.

Example 5-5. Given system M6 with & and + defined by table 25, system M6 can be represented as a lattice as shown in figure 16.

Table 25

Conjunction and Disjunction in M6

&	T	F	U	K1	K0	K
T	T	F	U	K1	K0	K
F	F	F	F	F	F	F
U	U	F	U	U	K0	K0
K1	K1	F	U	K1	K0	K
K0	K0	F	K0	K0	K0	K0
K	K	F	K0	K	K0	K

+	T	F	U	K1	K0	K
T	T	T	T	T	T	T
F	T	F	U	K1	K0	K
U	T	U	U	K1	U	K1
K1	T	K1	K1	K1	K1	K1
K0	T	K0	U	K1	K0	K
K	T	K	K1	K1	K	K

$$\begin{tikzcd}
 & T \arrow[d, dash] & \\
 & K1 \arrow[dl, dash] \arrow[dr, dash] & \\
U \arrow[dr, dash] & & K \arrow[dl, dash] \\
 & K0 \arrow[d, dash] & \\
 & F &
\end{tikzcd}$$

Figure 16. M6 Represented as a Lattice.

Example 5-6. Given conjunction and disjunction functions in Bochvar's B3, the logic values can not be represented as a lattice. As can be seen in table 26, there are two possible orderings. T-F-I is the order when glb defines conjunction and I-T-F is the order when lub defines disjunction.

Table 26

Conjunction and Disjunction in B3

&	T	I	F
T	T	I	F
I	I	I	I
F	F	I	F

+	T	I	F
T	T	I	T
I	I	I	I
F	T	I	F

5.4 The CLASSIFY Program.

The LISP program CLASSIFY finds the classification of a Designated Logic System within the mathematical hierarchy defined in this chapter. First, CLASSIFY attempts to classify the logic as an ordered designated system represented as a lattice. If this is possible, CLASSIFY will print the order from the 'least true' to the 'most true' elements. If the logic can't be classes as a lattice, CLASSIFY attempts to find a partial order of the logic elements. Failing this, CLASSIFY checks for identity elements and associative conjunction. If the logic passes these tests it is a designated monoid. If all these attempts fail, CLASSIFY defines the logic as a designated algebra.

```
> (classify)

Enter logic name: > sixval
; loading "b:SIXVAL.lsp"

Designated values: (T K1)
Neutral values: (U K)
```

```
Antidesignated values: (F K0)

Elements form a lattice.
(F) <= (K0) <= (U K) <= (K1) <= (T)
NIL
>
> (classify)

Enter logic name: > bochvar
; loading "b:BOCHVAR.lsp"

Designated values: (T)
Neutral values: (I)
Antidesignated values: (F)

Elements form a poset.
(I) <= (F) <= (T)
NIL
>
> (classify)

Enter logic name: > cyc4
; loading "b:CYC4.lsp"

Designated values: NIL
Neutral values: (3 2 1 4)
Antidesignated values: NIL

Logic is a designated monoid with identities: (3 1)
NIL
>
> (classify)

Enter logic name: > p4
; loading "b:P4.lsp"

Designated values: NIL
Neutral values: (3 2 1 4)
Antidesignated values: NIL

Logic is a designated algebra.
NIL
```

>
>

5.5 Summary.

The Designated Logic Systems can be placed in an hierarchy of mathematical systems with ever increasing constraints. Every Designated Logic System is also a designated algebra, but not all are designated monoids (e.g. systems with no supremum and systems with non-associative conjunction). Not all designated monoids define a partial order and not all partial orders can be represented as a lattice. Any system whose conjunction function does not impose the same order as its disjunction function can not be represented as a lattice. Figure 17 shows the hierarchy and where some of the systems studied in this chapter belong.

Classical logic can be represented as a lattice where $F \leq T$. Thus, the further down in the hierarchy a designated logic system falls, the more closely it resembles classical logic.

The mathematical definitions used in this chapter may be found in the following references: Grimaldi 1985, Lipschutz 1976, Liu 1977, Preparata and Yeh 1973, Ross and Wright 1985 and Stanat and McAllister 1977.

$$\begin{tikzcd}
\text{DESIGNATED ALGEBRAS (P3,P4)} \arrow[d, "\text{+ associativity, + supremum}"] \\
\text{DESIGNATED MONOIDS (CYC4)} \arrow[d, "\text{+ partial order}"] \\
\text{ORDERED DESIGNATED SYSTEMS (POSET) (B3)} \arrow[d, "\text{+ disjunction order}"] \\
\text{ORDERED DESIGNATED SYSTEMS (LATTICE) (K3 and M6)}
\end{tikzcd}$$

Figure 17. A Hierarchy of Designated Logic Systems.

In the next chapter we will discuss homomorphisms between the Designated Logic Systems and three valued logics. Using these homomorphisms we can limit the number of conjunction forms possible at each level of the hierarchy.

CHAPTER 6

CONJUNCTION HOMOMORPHISMS IN THE DESIGNATED HIERARCHY

In the hierarchy developed in chapter five we used a series of constraints to define a broad classes of designated logic systems. These classes were the designated algebra, the designated monoid and the ordered designated system. We would like to know how many types of conjunction functions there are possible at each level of the hierarchy such that the functions agree with classical logic when only designated and antidesignated values are in question. This will give us an idea of the complexity imposed on logic systems as a result of the third (neutral) logic class.

6.1 Conjunction Classes in Designated Logic Systems.

Because logic systems may have a very large number of elements, we would like to be able to describe conjunction functions using the logic's partition classes rather than individual elements. We will define function && to be &&: {D,N,A} x {D,N,A} -> {{D,N,A}, {D,N}, {D,A}, {N,A}, {D}, {N}, {A}, Ø} such that if X && Y = Z, there is some element a in X and some element b in Y such that a & b is an element of z for all z in Z where & is conjunction in the logic system. For example, if we wanted to define the && function for M6 with conjunction given in table 25, we would begin by finding M6's class assignments. We know from the FIND-ASSIGN examples in chapter 4 that M6's assignments are D = {T, K1}, N = {U, K} and A = {F, K0}. To find the value of D && D in M6 we would list all classes in which the results of a & b fall where a, b are elements of D. We see the following results:

T & T = T	T is an element of D
T & K1 = K1	K1 is an element of D
K1 & T = K1	K1 is an element of D
K1 & K1 = K1	K1 is an element of D

Therefore, D && D = {D} in M6.

Likewise the value of N && N in M6 may be found using the same procedure.

K & K = K	K is an element of N
K & U = K0	K0 is an element of A
U & K = K0	K0 is an element of A
U & U = U	U is an element of N

Therefore N && N = {N, A} in M6.

Table 27 is a representation of function && where P is the power set of {D,N,A}. In general, there are 9^8 possible functions since && can result in any element of {D,N,A}'s power set for any pair from the set {D,N,A}.

Table 27

A Meta-Conjunction Table For Classes D, N and A

&&	D	N	A
D	P	P	P
N	P	P	P
A	P	P	P

When we limit the functions in question to those which agree with classical logic when only designated and antidesignated values are in question, there are only 7^5 possible functions as seen in table 28 where P-Ø represents an element from the the power set of {D, N, A} not including the null (or empty) set.

Table 28

Function && With Classical Agreement

&&	D	N	A
D	{D}	P-Ø	{A}
N	P-Ø	P-Ø	P-Ø
A	{A}	P-Ø	{A}

6.2 Conjunction in Designated Algebras.

Given a designated algebra, we can limit the possible conjunction forms. In a designated algebra if a & b is designated then both a is designated and b is designated. As shown in table 29, there are only 3^5 possible functions where **NA** = {{N,A} {N} {A}}.

Table 29

&& in a Designated Algebra

&&	D	N	A
D	{D}	**NA**	{A}
N	**NA**	**NA**	**NA**
A	{A}	**NA**	{A}

6.3 Conjunction in a Designated Monoid.

A designated monoid has at least one identity element and an associative operator &. We can use the definition of a **designated** monoid to limit the number of conjunction forms.

Given a designated monoid, the existence of an identity **element allows** us to limit the possible conjunction forms to those shown in table 30 where $\square$ represents an element from the set {{N}, {N, A}}. Assume we have the identity element s in D. Since s & x = x' for all x in the logic where x and x' are in the same cycle-class, D && N and N && D can take a value from the list ({N}, {N,A}). Note there are now a total of $3^3 \times 2^2$ possible functions.

Table 30

&& When Conjunction Has an Identity Element

&&	D	N	A
D	{D}	□	{A}
N	□	NA	NA
A	{A}	NA	{A}

When we include the associative property of designated monoids as an additional constraint, we can limit the function patterns again. Assume N && N = {A}. Then we know (N && N) && N = N && (N && N), so A && N = N && A.

Thus, in a designated monoid we have three cases: N && N = {N}, N && N = {A} and N && N = {N, A}.

When N && N = A there are only the $2^2 + 2^2 + 2^2 = 12$ possible functions we see in table 31.

Table 31

&& in a Designated Monoid When N && N = {A}

&&	D	N	A
D	{D}	□	{A}
N	□	{A}	{N}
A	{A}	{N}	{A}

&&	D	N	A
D	{D}	□	{A}
N	□	{A}	{NA}
A	{A}	{NA}	{A}

&&	D	N	A
D	{D}	□	{A}
N	□	{A}	{A}
A	{A}	{A}	{A}

Otherwise there are 3^2 x 2^3 possible functions since D && N and N && D can be an element from {{N}, {N, A}}, N && A and A && N can be an element from {{N}, {A}, {N, A}} and N && N can be either {A, N} or {N}. There are, therefore, $3^3 * 3^2 = 72$ conjunction forms when N && N is not {A}. Thus, there are 72 + 12 = 84 possible conjunction forms which both agree with classical logic when only designated and antidesignated valued are considered and are designated monoids.

6.4 Conjunction in an Ordered Designated System.

An ordered designated system has several properties we can make use of$_4$ when determining the number of possible conjunction forms. Some of these properties are reflexivity, commutativity and the existence of the infimum.

Since all ordered designated systems are designated monoids we know there are at most 84 possible conjunction forms for ordered designated systems. We can use the reflexive property (x & x = x) of ordered designated systems to limit the number of possible conjunction forms by noting N && N ≠ {A} since there will be at least one instance when the conjunction of two neutral values will be neutral. This leaves us with 72 possible conjunction functions as described in the previous section.

Because the glb function is unique for any pair of elements in the ordered designated system, we know conjunction is commutative. We can use this property to limit the number of possible conjunction functions. Since D && N = N && D can be either {N} or {N, A}, N && A = A && N can be either {N}, {A} or {N, A} and N && N can be either {N} or {N, A} there are 2 * 3 * 2 = 12 possible conjunction forms for an ordered designated system.

When the infimum is in N there are eight possible functions as there are eight functions when the infimum is in A. Of these sixteen possible functions, four are the same, so we end up with the twelve conjunction types shown in table 32 (a) - (l).

Table 32

&& in Ordered Designated Systems

&&	D	N	A
D	{D}	{N}	{A}
N	{N}	{N}	{A}
A	{A}	{A}	{A}

(a)

&&	D	N	A
D	{D}	{NA}	{A}
N	{NA}	{N}	{A}
A	{A}	{A}	{A}

(b)

&&	D	N	A
D	{D}	{N}	{A}
N	{N}	{NA}	{A}
A	{A}	{A}	{A}

(c)

&&	D	N	A
D	{D}	{NA}	{A}
N	{NA}	{NA}	{A}
A	{A}	{A}	{A}

(d)

&&	D	N	A
D	{D}	{N}	{A}
N	{N}	{N}	{NA}
A	{A}	{NA}	{A}

(e)

&&	D	N	A
D	{D}	{NA}	{A}
N	{NA}	{N}	{NA}
A	{A}	{NA}	{A}

(f)

&&	D	N	A
D	{D}	{NA}	{A}
N	{NA}	{NA}	{NA}
A	{A}	{NA}	{A}

(g)

&&	D	N	A
D	{D}	{N}	{A}
N	{N}	{NA}	{NA}
A	{A}	{NA}	{A}

(h)

&&	D	N	A
D	{D}	{N}	{A}
N	{N}	{N}	{N}
A	{A}	{N}	{A}

(i)

&&	D	N	A
D	{D}	{N}	{A}
N	{N}	{NA}	{N}
A	{A}	{N}	{A}

(j)

&&	D	N	A
D	{D}	{NA}	{A}
N	{NA}	{N}	{N}
A	{A}	{N}	{A}

(k)

&&	D	N	A
D	{D}	{NA}	{A}
N	{NA}	{NA}	{N}
A	{A}	{N}	{A}

(l)

We can find a homomorphism between any ordered designated system and one of these twelve functions when conjunction agrees with classical logic in the designated and antidesignated cases. These twelve homomorphisms describe those logic systems which resemble classical logic most closely.

6.5 Examples.

Example 6-1. Kleene's logic is homomorphic to the conjunction form in table 32 (a).

Example 6-2. Bochvar's logic is homomorphic to the conjunction form in table 32 (i).

Example 6-3. M6 is homomorphic to the conjunction form in table 32 (c).

6.6 Summary.

We can find the conjunction form of those designated logic systems most like classical logic using a homomorphism between the classes of the logic partition and the classes to which conjunction maps. Those logics most like classical logic behave identically to classical logic when no neutral values are conjoined.

At each level of the designated hierarchy developed in the last chapter we can specify the number of conjunction forms which are closely related to classical logic. Using the constraints of the algebraic structures in the hierarchy we prune the number of possible functions from 3^5 functions in a designated algebra to twelve in an ordered designated system.

CHAPTER 7

CONCLUSION

In this work we have enhanced our understanding of the non-classical logics by providing a survey of some of the more important systems, defining designated logic systems, developing an algorithm for partitioning these systems and a classification scheme based on shared mathematical properties and similarities to classical logic.

In order to classify the many-valued logic systems we needed to understand their structure. We built an algorithm for defining a partition of logic values into three classes: designated (or truth-like) values, antidesignated (or false-like) values and neutral values. By allowing tautologies to take either designated or neutral values, and by allowing contradictions to take either antidesignated or neutral values, we built a system for which many of the tautologies and contradictions in classical logic hold in the designated logic systems as well.

We defined a number of criteria for logic systems in the form of constraints imposed on their complement and conjunction functions. We called logic systems meeting these constraints

Designated Logic Systems.

Using these constraints we developed an algorithm for finding all possible partitions of logic values into classes D (designated), N (neutral) and A (antidesignated). Thus, for any algebraic system having a permutation and binary function, we can find a classification of its elements such that the algebraic system is a Designated Logic System.

The hierarchy of Designated Logic Systems defined classes of logic systems based on shared mathematical properties. At the highest level of the hierarchy, the lattice, the Designated Logic Systems closely resemble classical logic. This hierarchy, together with the homomorphisms, show us there are relatively few forms of Designated Logic Systems which closely resemble classical logic. If the Designated Logic System agrees with classical logic whenever only designated and antidesignated values are in question, and if the logic's elements fall into the lattice structure in the designated hierarchy, there are only twelve possible conjunction forms the logic can have regardless of the number of its elements.

Although we do not assert that our classification scheme and definitions presented in this paper are the only correct method of defining and classifying Designated Logic Systems, we have developed a methodology which will enable any algebraic system to be transformed into a Designated Logic Systems given a permutation and a binary function.

This classification of logic systems opens many questions for further research: What is the smallest number of logic values required in each of the 12 classes? What happens to the classes when the definition of conjunction is changed (made more or less constrained)? What is the relationship between each of the classes? These questions, if researched, may provide further insight into the nature of Designated Logic Systems.

Any truthful study of the human reasoning process will show 'not all things are either strictly true nor strictly false.' People think, make decisions and correlate data and events with uncertain and incomplete data. To understand human reasoning we need non-classical logic tools. We hope this paper will provide a basis for an understanding of the algebraic nature of these systems.

APPENDIX 1

LOGIC SYSTEM DEFINITIONS

Classical Logic and Deviant Logic Definitions.

All logics are defined for NEGATION (~), CONJUNCTION (&), DISJUNCTION (+), IMPLICATION (=>) and EQUIVALENCE (<=>).

CLASSICAL LOGIC

~	
T	F
F	T

&	T	F
T	T	F
F	F	F

+	T	F
T	T	T
F	T	F

=>	T	F
T	T	F
F	T	T

<=>	T	F
T	T	F
F	F	T

THE THREE VALUED LOGIC OF LUKASIEWICZ (L3)

~	
T	F
I	I
F	T

&	T	I	F
T	T	I	F
I	I	I	F
F	F	F	F

+	T	I	F
T	T	T	T
I	T	I	I
F	T	I	F

=>	T	I	F
T	T	I	F
I	T	T	I
F	T	T	T

<=>	T	I	F
T	T	I	F
I	I	T	I
F	F	I	T

[Lukasiewicz, 1920]

THE FOUR VALUED SYSTEM OF LUKASIEWICZ

All logic values are of the form <a,b> where a and b are elements of {T,F}. Assume AND, OR, IMPLY, EQUAL and NOT are conjunction, disjunction, implication, equivalence and negation as defined in classical logic. The values of each function are

~<a,b> = <NOT(a),NOT(b)>
<a,b> & <c,d> = <AND(a,c),AND(b,d)>
<a,b> + <c,d> = <OR(a,c),OR(b,d)>
<a,b> => <c,d> = <IMPLY(a,c),IMPLY(b,d)>
<a,b> <=> <c,d> = <EQUAL(a,c),EQUAL(b,d)>

THE THREE VALUED SYSTEM OF BOCHVAR

~	
T	F
I	I
F	T

&	T	I	F
T	T	I	F
I	I	I	I
F	F	I	F

+	T	I	F
T	T	I	T
I	I	I	I
F	T	I	F

=>	T	I	F
T	T	I	F
I	I	I	I
F	T	I	T

<=>	T	I	F
T	T	I	F
I	I	I	I
F	F	I	T

[Bochvar, 1939]

THE THREE VALUED SYSTEM OF KLEENE

~	
T	F
I	I
F	T

&	T	I	F
T	T	I	F
I	I	I	F
F	F	F	F

+	T	I	F
T	T	T	T
I	T	I	I
F	T	I	F

=>	T	I	F
T	T	I	F
I	T	I	I
F	T	T	T

<=>	T	I	F
T	T	I	F
I	I	I	I
F	F	I	T

THE n VALUED SYSTEMS OF POST

Post defines a logic system of n values as:

~p = (p+1) mod n (here '+' denotes integer addition)
(p + q) = min(p,q)
(p & q) = ~(~p + ~q)

$$(p \Rightarrow q) = \neg p + q$$
$$(p \Leftrightarrow q) = (p \Rightarrow q) \,\&\, (q \Rightarrow p)$$

[Post, 1921]

POST'S THREE VALUED SYSTEM (P3)

~	
1	2
2	3
3	1

&	1	2	3
1	3	3	2
2	3	1	2
3	2	2	2

+	1	2	3
1	1	1	1
2	1	2	2
3	1	2	3

=>	1	2	3
1	1	2	2
2	1	2	3
3	1	1	1

<=>	1	2	3
1	3	3	3
2	3	1	2
3	3	2	3

MOUSSAVI'S SIX-VALUED LOGIC

K = {T,F,U}
K0 = {T,U}
K1 = {F,U}

~	
T	F
F	T
U	U
K1	K0
K0	K1
K	K

&	T	F	U	K1	K0	K
T	T	F	U	K1	K0	K
F	F	F	F	F	F	F
U	U	F	U	U	K0	K0
K1	K1	F	U	K1	K0	K
K0	K0	F	K0	K0	K0	K0
K	K	F	K0	K	K0	K

+	T	F	U	K1	K0	K
T	T	T	T	T	T	T
F	T	F	U	K1	K0	K
U	T	U	U	K1	U	K1
K1	T	K1	K1	K1	K1	K1
K0	T	K0	U	K1	K0	K
K	T	K	K1	K1	K	K

=>	T	F	U	K1	K0	K
T	T	F	U	K1	K0	K
F	T	T	T	T	T	T
U	T	U	U	K1	U	K1
K1	T	K0	U	K1	K0	K
K0	T	K1	K1	K1	K1	K1
K	T	K	K1	K1	K	K

[Moussavi, 1986]

APPENDIX 2

THE CREATE-LOGIC PROGRAM

The following LISP program is the source code used to create logic systems. Logic systems defined using this program are in the correct format for use with the FIND-CLASS and ORDER programs.

```
;                              CREATE-LOGIC
;
;The following functions query a user for a logic name and logic
;functions. It then creates an output file and defines the
;logic's functions and logical terms. After all functions for
;the new logic have been defined, these functions, along with
;a comprehensive list of logic values, is written to the file
;b:<logic name>.lsp.
;
;
;CREATE-LOGIC generates a logic system from scratch
;Input: none.
;Output: a logic system of terms and functions written to the
;file b:<logic name>.lsp
;
(defun create-logic ()
(progn
(terpri)
(princ "Enter logic name: ")
(setq lname (read))
(setq lname (strcat "b:" (symbol-name lname) ".lsp"))
(setq ofile (openo lname))
(setq flist nil)                  ;initialize list of functions
(setq vlist nil)                  ;initialize list of values
(getfun ofile)                    ;generate the logic functions
(printvals vlist ofile)))         ;save the functions to disk
```

```
;
;GETFUN gets individual logic functions from the user. A typical
;logic definition might include definitions for conjunction,
;disjunction, negation and implication. Functions must be input
;in the form:        (<fname> ((val1 [val2]) result)
;                             [(val1 [val2]) result)])
;For example, AND in classical logic would be defined as
;(AND ((T T) T)
;     ((T F) F)
;     ((F T) F)
;     ((F F) F))
;
;Input:
;   ofile: the output file to which the logic functions will
;   be written.
;Output: a series of logic functions and a list of function
;   names as well as a list of logic values.
;
(defun getfun (ofile)
(progn
(terpri)
(princ "Enter logic function: ")
(terpri)
(setq logfun (read))                        ;read the next function
(cond
((null logfun) (printf flist ofile))        ;if no function, write
(t (progn                                   ;results to output file
   (writef logfun)                          ;otherwise extract new
   (setq vlist (writevals (cdr logfun) nil))
   (writefun logfun ofile)                  ;logic values and go
   (getfun ofile))))))                      ;back for more functions

;WRITEF generates the function list for the logic. If you define
;a logic using ^,v and ~, WRITEF will form a list of those three
;functions along with the number of arguments each take.
;Input:
;   logfun: a logic function which needs to be added to the
;           function list (flist).
;Returns: the updated function list.
;
(defun writef (logfun)
(setq flist (cons (list (car logfun) (length (caadr logfun)))
                    flist)))

;PRINTF prints the function list to the output file. This creates
;a permanent record of all functions defined for the logic being
;created.
;Input:
;   flist: a list of function-name function-degree pairs.
;   ofile: the file name to which the list will be printed.
;Returns: nothing.
;
(defun printf (flist ofile)
(progn
   (terpri ofile)
   (princ "(setq flist '" ofile)     ;when the logic is loaded
   (prin1 flist ofile)               ;into the LISP environment,
   (princ ")" ofile)                 ;flist be a complete list of
   (terpri ofile)))                  ;all defined functions

;WRITEVALS collects and generates a list of logic values for the
;logic system. For example, in classical logic there are two
;logic values: true and false. In L3 there are three: T, I and
;F.
;Input:
;   logfun: a newly defined logic function
;   vlist: the current list of logic values
;Returns: the updated vlist. Any logic value refered to in logfun
;   but not contained in vlist will be added to vlist.
;
(defun writevals (logfun vlist)
(cond
((null logfun) vlist)
((listp (car logfun))
        (writevals (cdr logfun) (writevals (car logfun) vlist)))
((not (member (car logfun) vlist)) (cons (car logfun) vlist))
(t vlist)))

;PRINTVALS  prints  vlist  to the output  file.  This  creates  a
;permanant record of all logic values for the logic.
;Input:
;   vlist: the list of logic values in a logic system
;   ofile: the output file to which vlist will be written
;Returns: nothing
```

```
;
(defun printvals (vlist ofile)
(cond
((null vlist) (close ofile))
(t (progn
   (terpri ofile)
   (princ "(setq vlist '" ofile)  ;vlist is a list of all defined
   (prin1 vlist ofile)            ;logic values for a specific
   (princ  ")"  ofile)            ;system.
   (terpri ofile)
   (close ofile)))))

;WRITEFUN writes the function definition to the output file.
;Input:
;   logfun: the logic function to be defined.
;   ofile: the output file to which the logic function will
;   be written.
;Example:
;   If logfun were
;   (~ ((t) f)
;      ((f) t))
;   the following would be written to ofile:
;   (defun ~ (x)
;   (cond ((equal x '(t)) 'f)
;         ((equal x '(f)) 't)))
;Returns: nothing.
;
(defun writefun (logfun ofile)
(cond
((null logfun) nil)              ;done writing functions
(t (progn
   (terpri ofile)
   (princ "(defun " ofile)       ;start definition "(defun fn ...
   (prin1 (car logfun) ofile)
   (princ " (x) (cond " ofile)   ;write conditional statement
   (terpri ofile)
   (genfun (cdr logfun) ofile)   ;enter each function pair
   (princ "))" ofile)            ;then close fn and cond parens
   (terpri ofile)))))

;GENFUN generates the 'equal' parts of the logic functions.
;Input:
```

```
;   logfun: the logic function to be printed
;   ofile: the output file to which the logic function will
;   be printed.
;Example:
;   If logfun were ((t f) t) ... ) the following line would be
;   printed by a single iteration of genfun:
;   ((equal x '(t f)) t)
;   and genfun would be called to operate on the cdr of logfun.
;Returns: nil
;
(defun genfun (logfun ofile)
(cond
((null logfun) nil)                ;generated all function pairs
(t (progn
   (princ "((equal x '" ofile)
   (prin1 (caar logfun) ofile)
   (princ ") '" ofile)
   (prin1 (cadar logfun) ofile)
   (princ ")" ofile)
   (terpri ofile)
   (genfun (cdr logfun) ofile)))))
```

The following examples show the results of defining various logic systems using program CREATE-LOGIC.

CLASSICAL LOGIC

```
(defun ~ (x) (cond
((equal x '(T)) 'F)
((equal x '(F)) 'T)
))

(defun ^ (x) (cond
((equal x '(T T)) 'T)
((equal x '(T F)) 'F)
((equal x '(F T)) 'F)
((equal x '(F F)) 'F)
))

(defun V (x) (cond
((equal x '(T T)) 'T)
((equal x '(T F)) 'T)
((equal x '(F T)) 'T)
((equal x '(F F)) 'F)
))

(defun => (x) (cond
((equal x '(T T)) 'T)
((equal x '(T F)) 'F)
((equal x '(F T)) 'T)
((equal x '(F F)) 'T)
))

(setq flist '((=> 2) (V 2) (^ 2) (~ 1)))

(setq vlist '(F T))
```

LUKASIEWICZ'S 3-VALUED LOGIC

```
(defun ~ (x) (cond
((equal x '(T)) 'F)
((equal x '(I)) 'I)
((equal x '(F)) 'T)
))

(defun ^ (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'I)
((equal x '(T F)) 'F)
((equal x '(I T)) 'I)
((equal x '(I I)) 'I)
((equal x '(I F)) 'F)
((equal x '(F T)) 'F)
((equal x '(F I)) 'F)
((equal x '(F F)) 'F)
))

(defun V (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'T)
((equal x '(T F)) 'T)
((equal x '(I T)) 'T)
((equal x '(I I)) 'I)
((equal x '(I F)) 'I)
((equal x '(F T)) 'T)
((equal x '(F I)) 'I)
((equal x '(F F)) 'F)
))

(defun => (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'I)
((equal x '(T F)) 'F)
((equal x '(I T)) 'T)
((equal x '(I I)) 'T)
((equal x '(I F)) 'I)
((equal x '(F T)) 'T)
((equal x '(F I)) 'T)
((equal x '(F F)) 'T)
))

(defun <=> (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'I)
((equal x '(T F)) 'F)
((equal x '(I T)) 'I)
((equal x '(I I)) 'T)
((equal x '(I F)) 'I)
((equal x '(F T)) 'F)
((equal x '(F I)) 'I)
((equal x '(F F)) 'T)
))

(setq flist '(((<=> 2) (=> 2) (V 2) (^ 2) (~ 1)))

(setq vlist '(F I T))
```

KLEENE'S 3-VALUED LOGIC

```
(defun ~ (x) (cond
((equal x '(T)) 'F)
((equal x '(I)) 'I)
((equal x '(F)) 'T)
))


(defun ^ (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'I)
((equal x '(T F)) 'F)
((equal x '(I T)) 'I)
((equal x '(I I)) 'I)
((equal x '(I F)) 'F)
((equal x '(F T)) 'F)
((equal x '(F I)) 'F)
((equal x '(F F)) 'F)
))


(defun V (x) (cond
((equal x '(T T)) 'T)
((equal x '(I T)) 'T)
((equal x '(F T)) 'T)
((equal x '(T I)) 'T)
((equal x '(I I)) 'I)
((equal x '(F I)) 'I)
((equal x '(T F)) 'T)
((equal x '(I F)) 'I)
((equal x '(F F)) 'F)
))


(defun => (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'I)
((equal x '(T F)) 'F)
((equal x '(I T)) 'T)
((equal x '(I I)) 'I)
((equal x '(I F)) 'I)
((equal x '(F T)) 'T)
((equal x '(F I)) 'T)
((equal x '(F F)) 'T)
))
```

```
(defun <=> (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'I)
((equal x '(T F)) 'F)
((equal x '(I T)) 'I)
((equal x '(I I)) 'I)
((equal x '(I F)) 'I)
((equal x '(F T)) 'F)
((equal x '(F I)) 'I)
((equal x '(F F)) 'T)
))

(setq flist '(((<=> 2) (=> 2) (V 2) (^ 2) (~ 1)))

(setq vlist '(T I F))
```

POST'S 3-VALUED LOGIC

```
(defun ~ (x) (cond
((equal x '(1)) '2)
((equal x '(2)) '3)
((equal x '(3)) '1)
))

(defun V (x) (cond
((equal x '(1 1)) '1)
((equal x '(1 2)) '1)
((equal x '(1 3)) '1)
((equal x '(2 1)) '1)
((equal x '(2 2)) '2)
((equal x '(2 3)) '2)
((equal x '(3 1)) '1)
((equal x '(3 2)) '2)
((equal x '(3 3)) '3)
))

(defun ^ (x) (cond
((equal x '(1 1)) '3)
((equal x '(1 2)) '3)
((equal x '(1 3)) '2)
((equal x '(2 1)) '3)
((equal x '(2 2)) '1)
((equal x '(2 3)) '2)
((equal x '(3 1)) '2)
((equal x '(3 2)) '2)
((equal x '(3 3)) '2)
))

(defun => (x) (cond
((equal x '(1 1)) '1)
((equal x '(1 2)) '2)
((equal x '(1 3)) '2)
((equal x '(2 1)) '1)
((equal x '(2 2)) '2)
((equal x '(2 3)) '3)
((equal x '(3 1)) '1)
((equal x '(3 2)) '1)
((equal x '(3 3)) '1)
))

(defun <=> (x) (cond
((equal x '(1 1)) '3)
((equal x '(1 2)) '3)
((equal x '(1 3)) '3)
((equal x '(2 1)) '3)
((equal x '(2 2)) '1)
((equal x '(2 3)) '2)
((equal x '(3 1)) '3)
((equal x '(3 2)) '2)
((equal x '(3 3)) '3)
))

(setq flist '(((<=> 2) (=> 2) (^ 2) (V 2) (~ 1)))

(setq vlist '(2 3 1))
```

BOCHVAR'S 3-VALUED LOGIC

```
(defun ~ (x) (cond
((equal x '(T)) 'F)
((equal x '(I)) 'I)
((equal x '(F)) 'T)
))

(defun ^ (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'I)
((equal x '(T F)) 'F)
((equal x '(I T)) 'I)
((equal x '(I I)) 'I)
((equal x '(I F)) 'I)
((equal x '(F T)) 'F)
((equal x '(F I)) 'I)
((equal x '(F F)) 'F)
))

(defun V (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'I)
((equal x '(T F)) 'T)
((equal x '(I T)) 'I)
((equal x '(I I)) 'I)
((equal x '(I F)) 'I)
((equal x '(F T)) 'T)
((equal x '(F I)) 'I)
((equal x '(F F)) 'F)
))

(defun => (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'I)
((equal x '(T F)) 'F)
((equal x '(I T)) 'I)
((equal x '(I I)) 'I)
((equal x '(I F)) 'I)
((equal x '(F T)) 'T)
((equal x '(F I)) 'I)
((equal x '(F F)) 'T)
))

(defun <=> (x) (cond
((equal x '(T T)) 'T)
((equal x '(T I)) 'I)
((equal x '(T F)) 'F)
((equal x '(I T)) 'I)
((equal x '(I I)) 'I)
((equal x '(I F)) 'I)
((equal x '(F T)) 'F)
((equal x '(F I)) 'I)
((equal x '(F F)) 'T)
))

(setq flist '((<=> 2) (=> 2) (V 2) (^ 2) (~ 1)))

(setq vlist '(F I T))
```

MOUSSAVI'S 6-VALUED LOGIC

```
(defun ~ (x) (cond
((equal x '(T)) 'F)
((equal x '(F)) 'T)
((equal x '(U)) 'U)
((equal x '(K1)) 'K0)
((equal x '(K0)) 'K1)
((equal x '(K)) 'K)
))

(defun ^ (x) (cond
((equal x '(T T)) 'T)
((equal x '(T F)) 'F)
((equal x '(T U)) 'U)
((equal x '(T K1)) 'K1)
((equal x '(T K0)) 'K0)
((equal x '(T K)) 'K)
((equal x '(F T)) 'F)
((equal x '(F F)) 'F)
((equal x '(F U)) 'F)
((equal x '(F K1)) 'F)
((equal x '(F K0)) 'F)
((equal x '(F K)) 'F)
((equal x '(U T)) 'U)
((equal x '(U F)) 'F)
```

```
((equal x '(U U)) 'U)
((equal x '(U K1)) 'U)
((equal x '(U K0)) 'K0)
((equal x '(U K)) 'K0)
((equal x '(K1 T)) 'K1)
((equal x '(K1 F)) 'F)
((equal x '(K1 U)) 'U)
((equal x '(K1 K1)) 'K1)
((equal x '(K1 K0)) 'K0)
((equal x '(K1 K)) 'K)
((equal x '(K0 T)) 'K0)
((equal x '(K0 F)) 'F)
((equal x '(K0 U)) 'K0)
((equal x '(K0 K1)) 'K0)
((equal x '(K0 K0)) 'K0)
((equal x '(K0 K)) 'K0)
((equal x '(K T)) 'K)
((equal x '(K F)) 'F)
((equal x '(K U)) 'K0)
((equal x '(K K1)) 'K)
((equal x '(K K0)) 'K0)
((equal x '(K K)) 'K)
))

(defun V (x) (cond
((equal x '(T T)) 'T)
((equal x '(T F)) 'T)
((equal x '(T U)) 'T)
((equal x '(T K1)) 'T)
((equal x '(T K0)) 'T)
((equal x '(T K)) 'T)
((equal x '(F T)) 'T)
((equal x '(F F)) 'F)
((equal x '(F U)) 'U)
((equal x '(F K1)) 'K1)
((equal x '(F K0)) 'K0)
((equal x '(F K)) 'K)
((equal x '(U T)) 'T)
((equal x '(U F)) 'U)
((equal x '(U U)) 'U)
((equal x '(U K1)) 'K1)
((equal x '(U K0)) 'U)
```

```
((equal x '(U K)) 'K1)
((equal x '(K1 T)) 'T)
((equal x '(K1 F)) 'K1)
((equal x '(K1 U)) 'K1)
((equal x '(K1 K1)) 'K1)
((equal x '(K1 K0)) 'K1)
((equal x '(K1 K)) 'K1)
((equal x '(K0 T)) 'T)
((equal x '(K0 F)) 'K0)
((equal x '(K0 U)) 'U)
((equal x '(K0 K1)) 'K1)
((equal x '(K0 K0)) 'K0)
((equal x '(K0 K)) 'K)
((equal x '(K T)) 'T)
((equal x '(K F)) 'K)
((equal x '(K U)) 'K1)
((equal x '(K K1)) 'K1)
((equal x '(K K0)) 'K)
((equal x '(K K)) 'K)
))

(defun => (x) (cond
((equal x '(T T)) 'T)
((equal x '(T F)) 'F)
((equal x '(T U)) 'U)
((equal x '(T K1)) 'K1)
((equal x '(T K0)) 'K0)
((equal x '(T K)) 'K)
((equal x '(F T)) 'T)
((equal x '(F F)) 'T)
((equal x '(F U)) 'T)
((equal x '(F K1)) 'T)
((equal x '(F K0)) 'T)
((equal x '(F K)) 'T)
((equal x '(U T)) 'T)
((equal x '(U F)) 'U)
((equal x '(U U)) 'U)
((equal x '(U K1)) 'K1)
((equal x '(U K0)) 'U)
((equal x '(U K)) 'K1)
((equal x '(K1 T)) 'T)
((equal x '(K1 F)) 'K0)
```

```
((equal x '(K1 U)) 'U)
((equal x '(K1 K1)) 'K1)
((equal x '(K1 K0)) 'K0)
((equal x '(K1 K)) 'K)
((equal x '(K0 T)) 'T)
((equal x '(K0 F)) 'K1)
((equal x '(K0 U)) 'K1)
((equal x '(K0 K1)) 'K1)
((equal x '(K0 K0)) 'K1)
((equal x '(K0 K)) 'K1)
((equal x '(K T)) 'T)
((equal x '(K F)) 'K)
((equal x '(K U)) 'K1)
((equal x '(K K1)) 'K1)
((equal x '(K K0)) 'K)
((equal x '(K K)) 'K)
))

(setq flist '((^ 2) (~ 1) (V 2) (=> 2)))

(setq vlist '(F K0 U K K1 T))
```

APPENDIX 3

THE FIND-CYCLES PROGRAM

As discussed in chapter three, program FIND-CYCLES finds all the odd and even cycles in a logic system using the complementation function. If the complement function is not a permutation, FIND-CYCLES returns an error message.

```
;
;                              FIND-CYCLES
;
;FIND-CYCLES finds all odd and even cycles in the complementation
;permutation of a logic. If the logic's negation function is not
;a permutation, find-cycles returns an error message. Otherwise
;find-cycles returns a list of even cycles and a list of odd
;cycles in the form ((v1 v2...vn)(v1 v2 ...vn)...) where each set
;of v1...vn values is one cycle.
;
;Input:
;   The logic whose cycles are to be found. The negation
;   function must be defined in the format created by the
;   program CREATE-LOGIC. The logic function file must be
;   b:<lname>.lsp where <lname> is the argument to FIND-
;   CYCLES.
;Returns:
;   A list of the odd and even cycles of the selected logic.
;
(defun find-cycles ()
(progn
 (terpri)
 (princ "Enter logic name: ")
 (load (strcat "b:" (symbol-name (read)) ".lsp"))
 (setq even nil)
 (setq odd nil)
 (cond
  ((not (permutationp vlist nil)) (progn
                                           (princ "not a permutation")
                                           (terpri)))
   (t (progn
       (cycle vlist)
       (list odd even))))))

;PERMUTATIONP determines if the complement function in the given
;logic is a permutation.
;Input:
;   ivals: the list of elements whose complements need to be
;   determined.
;   ovals: the list of elements the complement of some value
;   erstwhile in ivals. For example, if T was in ivals and ~T
;   = F then F will be in ovals.
;Returns:
;   T if the complement function is a permutation
;   nil if the complement function is not a permutation
(defun permutationp (ivals ovals)
(cond
 ((and (null ivals) (same-list ovals vlist)) t)
 ((null ivals) nil)
 ((member (~ (list (car ivals))) ovals) nil)
 (t (permutationp (cdr ivals) (cons (~ (list (car ivals)))
                                        ovals)))))

;CYCLE finds all cycles in a permutation and puts them in global
;lists odd and even.
;
;Input:
;   vals: the list of values in a permutation to find the
;   cycles for.
;Returns:
;   Nothing. Global lists odd and even are updated.
;
(defun cycle (vals)
(cond
 ((not (null vals))
  (cycle (save-cyc (find-cycle (list (car vals))) vals)))))

;SAVE-CYC simply stores the cycle in the correct global list
;and removes that cycle's elements from the vals list.
;
```

```
;Input:
;   cyc: the cycle to be stored
;   vals: the list of elements to update
;Returns:
;   The updated vals list
;
(defun save-cyc (cyc vals)
(prog2
 (if (evenp (length cyc))
     (setq even (cons cyc even))
     (setq odd (cons cyc odd)))
 (rem-cycle vals cyc)))

;FIND-CYCLE finds the remaining cycle of the elements in cyc.
;
;Input:
;   cyc: the list of values already in the cycle.
;Returns:
;   the entire cycle in reverse order
;
(defun find-cycle (cyc)
(cond
 ((member (~ (list (car cyc))) cyc :test equal) cyc)
 (t (find-cycle (cons (~ (list (car cyc))) cyc)))))

;REM-CYCLE removes all elements of list cyc from list vals
;
;Input:
;   vals: the input list to alter
;   cyc: the list of elements to remove from vals.
;Returns:
;   The updated vals list
;
(defun rem-cycle (vals cyc)
(cond
 ((null cyc) vals)
 (t (rem-cycle (remove (car cyc) vals) (cdr cyc)))))

;ADD-IF adds item to input list if it is not already a member.
;
;Input:
;   item: the item to add to l
;   l   : the list to be changed
;Returns:
;   The updated list.
;
(defun add-if (item l)
(cond
 ((member item l :test equal) l)
 (t (cons item l))))

;SAME-LIST return T if two lists have the same elements,
;else return nil.
;
;Input:
;   l1: one list to compare
;   l2: the other list
;Returns:
;   A boolean.
;
(defun same-list (l1 l2)
(cond
 ((null l1) (null l2))
 ((member (car l1) l2) (same-list (cdr l1)
                                  (remove (car l1) l2
                                   :test equal)))
(t nil)))

;SPLIT splits an even cycle into two halves. Each half
;will either be in the designated class or the antidesignated
;class of a designated partition of a logic's values.
;
;INPUT:
;   cyc - the cycle to split
;RETURNS:
;   A list of the two halves of the cycle. For example, if
;   the cycle is (v1 v2 v3 v4), SPLIT will return the list
;   ((v1 v3) (v2 v4)).
;
(defun split (cyc)
(cond
((null cyc) nil)
(t (expand-split (car cyc) (cadr cyc) (split (cddr cyc))))))
```

```
;EXPAND-SPLIT adds values to each half of a split cycle. For
;example, given (expand-split v1 v2 ((v3) (v4))), the function
;will return ((v1 v3) (v2 v4)).
;
;Input:
;   v1 - the value to add to the first half of the split.
;   v2 - the value to add to the second half of the split.
;   l  - the previously split values.
;Returns:
;   A new split list with v1 and v2 added.
;
(defun EXPAND-SPLIT (v1 v2 l)
(list (cons v1 (car l)) (cons v2 (cadr l))))

;GEN-CYCLE-CLASSES generates a list of all the cycle
;classes in the logic.
;
;Input:
;   evencyc: a list of all the even cycles.
;Returns:
;   A list of all the cycle-classes.
;
(defun gen-cycle-classes (evencyc)
(cond
 ((null evencyc) odd)
 (t (append (split (car evencyc))
            (gen-cycle-classes (cdr evencyc))))))
```

APPENDIX 4

THE FIND-ASSIGN PROGRAM

Program FIND-ASSIGN determines the assignment of logic values to partition classes using the algorithm developed in chapter four.

```
;                           FIND-ASSIGN
;
;This series of programs finds all possible assignments of logic
;elements to designated and antidesignated classes. We assume
;the conjunction function of the logic is defined as specified
;in the CREATE-LOGIC program and that it is loaded. FIND-ASSIGN
;finds all possible assignments of even cycles in the logic which
;result in well defined conjunction. If there are no such
;assignments, FIND-ASSIGN attempts to find the offending cycles
;and puts them into the neutral class.
;
(defun find-assign ()
(progn
 (setq possible nil)
 (setq D nil)
 (setq A nil)
 (if (not (boundp 'find-cycles)) (load 'b:cycle))
 (find-cycles)
 (setq N (flatten odd))
 (if (consp even) (test-assign even nil nil))
 (if (and (consp even) (null possible)) (find-fault))
 (if (and (consp even) (ccnsp possible))
     (do-assign (best-assign possible)))
 (terpri)
 (princ "Designated values: ")
 (prin1 D)
 (terpri)
 (princ "Neutral values: ")
 (prin1 N)
 (terpri)
 (princ "Antidesignated values: ")
 (prin1 A)
 (terpri)))

;FLATTEN takes a multi-level list and returns a flat list
;in its place.
;Input:
;   exp: the expression to flatten
;Returns:
;   The flat expression.
;
(defun flatten (exp)
(cond
 ((null exp) nil)
 ((atom exp) (list exp))
 (t (append (flatten (car exp)) (flatten (cdr exp))))))

;DO-ASSIGN makes the final assignment of values to the
;designated and antidesignated classes.
;
;Input:
;   alist: the assignment to make in the form ((D) (A))
;   where D is the designated values and A the antidesignated.
;Returns:
;   The sets D, A and N as globals.
;
(defun do-assign (alist)
(progn
 (setq D (car alist))
 (setq A (cadr alist))
 alist))


;TEST-ASSIGN finds all possible assignments of even cycles and
;tests them to find any resulting in well defined conjunction.
;All such assignments are stacked in the global location
;POSSIBLE.
;
;Input:
;   evencyc: a list of all the even cycles to be tested.
;   D-prime: a test assignment of designated values.
;   A-prime: a test assignment of antidesignated values.
;
;Returns:
;   possible: a list of all possible assignments in the
;   form (((desig) (antidesig)) ((desig) (antidesig)) ... )
;
(defun test-assign (evencyc D-prime A-prime)
(cond
 ((and (null evencyc) (testp D-prime (cddar (cddr (car ^))))))
  (setq possible (cons (list D-prime A-prime) possible)))
 ((not (null evencyc))
  (progn
   (test-assign (cdr evencyc) (append D-prime
                                       (class1 (car evencyc)))
                              (append A-prime
                                       (class2 (car evencyc))))
   (test-assign (cdr evencyc) (append D-prime
                                       (class2 (car evencyc)))
                              (append A-prime
                                       (class1 (car evencyc))))
   possible))))

;CLASS1 collects all the even numbered elements of an even
;cycle. For example, given a cycle (a b c d e f), Class1 would
;return (a c e).
;
;Input:
;   evencyc: the even cycle
;Returns:
;   The even half of the even cycle.
;
(defun class1 (evencyc)
(cond
((null evencyc) nil)
(T (cons (car evencyc) (class1 (cddr evencyc))))))

;CLASS2 selects the even numbered elements from an even cycle.
;
;Input:
;   evencyc: the even cycle to be split
;Returns:
;   The odd numbered elements of the even cycle.
```

```
;
(defun class2 (evencyc)
(cond
((null evencyc) nil)
(t (cons (cadr evencyc) (class2 (cddr evencyc))))))

;TESTP returns T if an assignment of elements to the designated
;and antidesignated classes results in well defined conjunction.
;Otherwise TESTP returns NIL.
;
(defun testp (desig conj)
(cond
((null desig) T)
((null conj) T)
((and
 (member (eval (cadar conj)) desig)
 (or
 (not (member (car (eval (caddr (caar conj)))) desig))
 (not (member (cadr (eval (caddr (caar conj)))) desig)))) nil)
(T (testp desig (cdr conj)))))

;BEST-ASSIGN selects the best possible assignment of designated
;and antidesignated values based on the names true, T, false and
;F. BEST-ASSIGN tries to put T or true in the designated class
;and false or F in the antidesignated class.
;
;Input:
;   plist: the list of possible assignments in the form
;   (((D) (A)) ((D) (A)) ... ((D) (A))).
;Returns:
;   The best possible assignment of values to the
;   designated and antidesignated classes.
;
(defun best-assign (plist)
(cond
 ((equal (length plist) 1) (car plist))
 ((and (member 'true vlist) (member 'false vlist))
       (find-both 'true 'false plist))
 ((and (member 'T vlist) (member 'F vlist))
       (find-both 'T 'F plist))
 ((member 'false vlist) (find-f nil 'false plist))
 ((member 'F vlist) (find-f nil 'F plist))
 ((member 'true vlist) (find-t 'true plist))
 ((member 'T vlist) (find-t 'T plist))
 (T (car plist))))

;FIND-BOTH attempts to find an assignment for which F or false
;is antidesignated and T or true is designated. Failing this,
;FIND-BOTH will try to find where F or false is antidesignated.
;
;Input:
;   desig: the default designated value of the system. This will
;   either be T or true.
;   antidesig: the default antidesignated value of the system.
;   This will be either F or false.
;   plist: the list of possible assignments.
;Returns:
;   a list of the form ((D) (A)) where D is the designated
;   values and A is the antidesignated values.
;
(defun find-both (desig antidesig plist)
(cond
 ((null plist) nil)
 ((and (member desig (caar plist)) (member antidesig
                                           (cadar plist)))
       (car plist))
 ((find-both desig antidesig (cdr plist)))
 (t (find-f desig antidesig plist))))

;FIND-F attempts to find an assignment where the default anti-
;designated value is assigned to the antidesignated class. If
;this fails, FIND-F attempts to put the default designated
;value in the designated class. If the default designated value
;is nil, there, neither T nor true are elements of the logic.
;
;Input:
;   desig: the default designated value of the logic system. It
;   will be either true or T.
;   antidesig: the default antidesignated value of the logic
;   system. It will be either F or false.
;   plist: the list of possible assignments.
;Returns:
;   A list of the form ((D) (A)) where D is the designated
;   values and A the antidesignated.
```

```
;
(defun find-f (desig antidesig plist)
(cond
 ((null plist) nil)
 ((member antidesig (cadar plist)) (car plist))
 ((find-f desig antidesig (cdr plist)))
 (t (find-t desig plist))))

;FIND-T will attempt to find an assignment of logic values to
;partition classes such that the default designated value of
;the system is in the designated class. If this fails, FIND-T
;will return the first assignment in the list.
;
;Input:
;   desig: the default designated value of the logic system.
;   plist: the list of possible assignments.
;Returns:
;   A list of the form ((D) (A)) where D is the designated
;   values and A the antidesignated.
;
(defun find-t (desig plist)
(cond
 ((null plist) nil)
 ((member desig (caar plist)) (car plist))
 ((find-t desig (cdr plist)))
 (t (car plist))))

;FIND-FAULT will find the elements of a logic's even cycles
;causing the logic to have poorly defined conjunction.
;FIND-FAULT will delete the fewest even cycles from the
;D-A classification and put them in N. FIND-FAULT first
;tries to find an assignment after deleting one even
;cycle from the even cycle list. Then two, three ...
;As a last resort, FIND-FAULT will put all logic elements
;in the neutral class.
;
;Input:
;   None although the global EVEN is used.
;Returns:
;   Updated globals EVEN, POSSIBLE and N reflecting the
;   new values in N removed from EVEN.
;
```

```
(defun find-fault ()
(cond
 ((equal (length even) 1) (adjust-even even))
 (T (adjust-even (delete-cycles 1 (C (length even) 1))))))

;C will find all combinations of R items out of a set of N
;distinct items. The formula for this is N!/(N-R)!R!. If N=R
;then C(N,R)=1, if N < R then there are no possible
;combinations. The problem is equivalent to having N balls and
;R boxes. Try to see how many different sets of balls will
;fill the boxes.
;
;Input:
;   N: the number of elements to choose from
;   R: the number of elements to choose.
;Returns:
;   A list of all possible choices.
;
(defun C (N R)
(cond
((< N R) nil)                          ;fewer balls than boxes
((= N R) (list (consec N)))            ;equal balls and boxes
((= R 1) (singleton N))                ;only one box
(t (append (app N (C (1- N) (1- R)))
           (C (1- N) R)))))

;CONSEC lists the integers from 1 to N if N is positive, else
;it returns nil. For example, (consec 3) would reture (1 2 3).
;
;Input:
;   N: the number of consecutive integers to list.
;Returns:
;   The list of consecutive integers.
;
(defun consec (N)
(cond
((<= N 0) nil)
((= N 1) '(1))
(t (cons N (consec (1- N))))))

;SINGLETON lists individual numbers from 1 to N in individual
;lists. For example, (singleton 3) would return ((1)(2)(3)).
```

```
;
;Input:
;   N: the number of singletons to create
;Returns:
;   The list of singleton lists.
;
(defun singleton (N)
(cond
((<= N 0) nil)
(t (cons (list N) (singleton (1- N))))))

;APP appends N to each list in L. For example, (app 4 '
;((3 2) (3 1) (2 1))) would return ((4 3 2)(4 3 1)(4 2 1)).
;
;Input:
;   N: The number to append to each list
;   L: The list to append to
;Returns:
;   A list of the appended lists.
;
(defun app (N L)
(cond
((zerop N) L)
((null L) nil)
(t (cons (cons N (car L)) (app N (cdr L))))))

;ADJUST-EVEN will remove elements from even cycles and place
;then in class N.
;
;Input:
;   l: the list of even cycles
;Returns:
;   Updated EVEN and N as globals.
;
(defun adjust-even (l)
(if (consp l) (progn
               (delete (car l) even :test equal)
               (setq N (append (car l) N))
               (adjust-even (cdr l)))))

;DELETE-CYCLES tries to find a possible assignment of
;elements to classes D and A by deleting even cycles
```

```
;and putting them in N. DELETE-CYCLES starts off deleting
;one cycle at a time, then two ... As a last resort,
;DELETE-CYCLES will put all elements in class N.
;
;Input:
;   num: the number of cycles to delete from even.
;   combos: all possible combinations of num elements
;   to be deleted from the even cycle list. For example,
;   if num is 2 and there are 4 even cycles, combos would
;   be the list ((1 2) (1 3) (1 4) (2 3) (2 4) (3 4)).
;   Every possible combintaion of two elements would be
;   removed from even until one combination was found to
;   result in well defined conjunction.
;Returns:
;   A list of the even cycles to be deleted from EVEN. The
;   elements from these same cycles will be added to class N.
;
(defun delete-cycles (num combos)
(cond
 ((equal num (length even)) even)
 ((null combos)
  (delete-cycles (1+ num) (C (length even) (1+ num))))
 ((test-assign (remove-cyc (car combos) even) nil nil)
              (list-cyc (car combos) even))
 (t (delete-cycles num (cdr combos)))))

;REMOVE-CYC removes the elements of l1 from l2.
;
;Input:
;   l1: the list of elements to remove
;   l2:the list to remove them from
;Returns:
;   The list l2 with elements from l1 removed.
;
(defun remove-cyc (l1 l2)
(cond
 ((null l1) l2)
 (T (remove-cyc (remove (car (last l1)) l1 :test equal)
               (remove (nth (1- (car (last l1))) l2)
                       l2 :test equal)))))

;LIST-CYC lists the elements of l2 numbered in l1 with
```

```
;a one origin.
;
;Input:
;   l1: the list of element numbers to list
;   l2: the list of elements to list
;Returns:
;   A list of the elements.
;
(defun list-cyc (l1 l2)
(cond
 ((null l1) nil)
 (T (cons (nth (- (car l1) 1) l2) (list-cyc (cdr l1) l2)))))
```

APPENDIX 5

THE CLASSIFY PROGRAM

Program CLASSIFY determines where in the hierarchy developed in chapter five logic systems fit. CLASSIFY tries in turn: lattice, poset, momoid and algebra. All logic systems default to algebra if they don't fall into any of the three other categories.

```
;                                CLASSIFY
;
;CLASSIFY determines where a logic fits into the hierarchy
;of designated logic systems. The logic may be a lattice, a
;poset, a monoid or an algebra. Classify also finds the
;designated and antidesignated values of a logic.
;
(defun classify ()
(progn
 (if (not (boundp 'find-assign)) (load 'b:assign))
 (find-assign)
 (setq even nil)
 (setq odd nil)
 (cycle vlist)
 (cond
  ((latticep) (progn
               (terpri)
               (princ "Elements form a lattice. ")
               (terpri)
               (print-order (orderup vlist (make-and-slist vlist
                            (init-slist vlist))))
               (terpri)))
  ((posetp (make-and-slist vlist (init-slist vlist)))
   (progn
    (terpri)
    (princ "Elements form a poset. ")
    (terpri)
    (print-order (orderup vlist
                     (make-and-slist vlist (init-slist vlist))))
    (terpri)))
  ((and (assocp vlist vlist vlist)
        (find-idents vlist (gen-cycle-classes even)))
   (progn
    (terpri)
    (princ "Logic is a designated monoid with identities: ")
    (print (find-idents vlist (gen-cycle-classes even)))
    (terpri)))
  (t (progn
      (terpri)
      (princ "Logic is a designated algebra." )
      (terpri))))))

;PRINT-ORDER will print out logic values in 'least true' to
;'most true' order.
;
;Input:
;   ord: a list of the ordered logic values
;Returns:
;   The list printed to default output sink
;
(defun print-order (ord)
(cond
 ((null (cdr ord)) (prin1 (car ord)))
 (t (progn (prin1 (car ord))
           (princ " <= ")
           (print-order (cdr ord))))))

;LATTICEP
;Given a set of logic values and the functions AND (^) and OR
;(v), These functions will find a possible ordering of the
;values.
;
;Order the current logic system using the current AND (^) and OR
;(v) functions. Returns the order of the arguments if there is
;one, else returns the possible orderings.
;
(defun latticep ()
(and (posetp (make-and-slist vlist (init-slist vlist)))
     (equal-order (orderup vlist
                  (make-and-slist vlist (init-slist vlist)))
                  (orderup vlist
                  (make-or-slist vlist (init-slist vlist))))))

;Determine if two orderings are identical.
;Input:
;   o1: an ordering of logic values of the form ((...)(...)(...))
;   o2: ditto
;Returns:
;   T if the orderings are identical, otherwise Nil
;
(defun equal-order (o1 o2)
(cond
((null o1) (null o2))
((same-list (car o1) (car o2)) (equal-order (cdr o1) (cdr o2)))
(t nil)))

;Order all elements in a logic system from the least to the most
;'true.' Logic elements on the same level will be listed
;together. Use the function AND to determine order.
;
;Input:
;   arglist: the list of logic values to order
;   slist:   a list of the form ((x1 (...))(x2 (...)) ... )
;            such that the xi are greater than or equal to any
;            element in their association list.
;Returns:
;   The ordering of the input argument list.
;
(defun orderup (arglist slist)
(cond
((null arglist) nil)
((equal (length arglist) 1) (list arglist))
(t (cons (setq min (find-min slist vlist (length vlist)))
         (orderup (new-arglist min arglist)
                  (new-slist min slist))))))

;make an association list of all logic values and a list of all
;other logic values less than it.
;
```

```
;Input:
;   args:  the list of logic values.
;   slist: An association list of the form
;
;     ((x1 ( ... ))
;      (x2 ( ... ))
;             .
;             .
;             .
;      (xn ( ... ))
;   as described above.
;   Returns:
;      The association list based on the function AND (^)
;
(defun make-and-slist (args slist)
(cond
((null args) slist)
(t (make-and-slist (cdr args)
                (make-and-slist2 (car args) vlist slist)))))

;find all possible values of (arg1 ^ arg2) and enter the resulst
;in the respective slists for each element
;
;Input:
;   arg1: The first argument for ANDing to every possible logic
;         value.
;   arg2: The list of all possible second arguments for ANDing.
;   slist: The list thus far of logic association lists
;Returns:
;   The slist updated for arg1.
;
(defun make-and-slist2 (arg1 arg2 slist)
(cond
((null arg2) slist)
(t (make-and-slist2 arg1 (cdr arg2)
   (add-arg (^ (list arg1 (car arg2))) (car arg2)
   (add-arg (^ (list arg1 (car arg2))) arg1 slist))))))

;Make a list of all values less than the current value using the
;function OR.
;ditto input and output for make-and-slist
;
```

```
(defun make-or-slist (args slist)
(cond
((null args) slist)
(t (make-or-slist (cdr args) (make-or-slist2 (car args)
                                             vlist slist)))))

;Find all values in arg2 less than or equal to arg1 and update
;slist to reflect these changes.
;Ditto input and output for make-and-slist2
;
(defun make-or-slist2 (arg1 arg2 slist)
(cond
((null arg2) slist)
(t (make-or-slist2 arg1 (cdr arg2)
                   (add-arg arg1 (V (list arg1 (car arg2)))
                   (add-arg (car arg2) (v (list arg1
                                               (car arg2)))
                                               slist))))))

;add arg1 to arg2's slist if it is not already a member
;Input:
;   arg1: the logic value to be added to arg2's association
;   list.
;   arg2: the logic value whos association list will be updated.
;   slist: the association list to be updated.
;Returns:
;   The updated slist.
;
(defun add-arg (arg1 arg2 slist)
(cond
((equal (caar slist) arg2)
 (cond
 ((member arg1 (cadar slist)) slist)
 (t (cons
    (list arg2 (cons arg1 (cadar slist)))
    (cdr slist)))))
(t (cons (car slist) (add-arg arg1 arg2 (cdr slist))))))

;initialize the slist to ((x1 ())
;                          (x2 ()) ... (xn ()))
;Input:
;   arglist: the list of logic values
```

```
;Returns:
;   The initialized association list.
;
(defun init-slist (arglist)
(cond
((null arglist) nil)
(t (cons (cons (car arglist) '(())) (init-slist
                                          (cdr arglist))))))

;return T if two lists have the same elements, else return nil
;
;Input:
;   l1: one list to compare
;   l2: the other list
;Returns:
;   A boolean.
;
(defun same-list (l1 l2)
(cond
((null l1) (null l2))
((member (car l1) l2) (same-list (cdr l1)
                                 (remove (car l1) l2
                                                :test equal)))
(t nil)))

;return the set of minimum values in a list.
;
;Input:
;   slist: the association list for each logic value.
;   minlist: the current list of minimal logic values.
;   len: the number of logic values 'less' than the current
;        minimal logic values.
;Returns:
;   The list of minimal logic values.
;
(defun find-min (slist minlist len)
(cond
((null slist) minlist)
((< (length (cadar slist)) len)
 (find-min (cdr slist) (list (caar slist))
                            (length (cadar slist))))
((equal len (length (cadar slist)))
 (find-min (cdr slist) (cons (caar slist) minlist) len))
(t (find-min (cdr slist) minlist len))))

;delete an association list entry from a list
;Input:
;   key: the logic value whos association list is to be deleted.
;   slist: the list of association lists.
;Returns:
;   The updated association list.
;
(defun delist (key alist)
(cond
((assoc key alist) (remove (assoc key alist) alist :test equal))
(t alist)))

;delete all association lists of minlist from slist
;return the new slist
;Input:
;   minlist: the list of minimal values whos association lists
;   are to be deleted.
;   slist: the list of association lists.
;
(defun new-slist (minlist slist)
(cond
((null minlist) slist)
(t (new-slist (cdr minlist) (delist (car minlist) slist)))))

;delete elements of minlist from arglist and return resulting
;arglist.
;
;Input:
;   minlist: the list of values to be deleted from the arglist.
;   arglist: the remaining list of logic values to be ordered.
;Returns:
;   The updated argument list.
;
(defun new-arglist (minlist arglist)
(cond
((null minlist) arglist)
(t (new-arglist (cdr minlist) (remove (car minlist) arglist
                                                      :test equal)))))
```

```
;ASSOCP determines if a conjunction function is associative
;by testing the equivalence of (a ^ b) ^ c with a ^ (b ^ c)
;for all possible a's b's and c's.
;
;Input:
;   avlist: the list of possible values for a
;   bvlist: the list of possible values for b
;   cvlist: the list of possible values for c
;Returns:
;   T if the conjunction function is associative and
;   nil otherwise.
;
(defun assocp (avlist bvlist cvlist)
(cond
 ((and (consp cvlist) (is-assoc (car avlist)
                               (car bvlist)
                               (car cvlist)))
  (assocp avlist bvlist (cdr cvlist)))
 ((consp cvlist) nil)
 ((and (null cvlist) (consp (cdr bvlist)))
  (assocp avlist (cdr bvlist) vlist))
 ((and (null cvlist) (null (cdr bvlist)) (consp (cdr avlist)))
  (assocp (cdr avlist) vlist vlist))
 (t t)))

;IS-ASSOC returns t if (a ^ b) ^ c = a ^ (b ^ c).
;
;Input:
;   a,b,c: the three logic values to test
;Returns:
;   T if they are associative, else nil.
;
(defun is-assoc (a b c)
(equal (^ (list (^ (list a b)) c))
       (^ (list a (^ (list b c))))))

;POSETP returns T if a logic can be represented as a poset,
;otherwise POSETP returns nil.
;
;Input:
;   slist: the association list of logic values to other
;   values <= to it.
```

```
;Returns:
;   T if the logic is a poset, else nil.
;
(defun posetp (slist)
(and (transitivep vlist slist)
     (reflexivep  slist)
     (antisymmetricp  vlist slist)))

;TRANSITIVEP returns T if the relation defined by conjunction
;is transitive. If a <= b and b <= c then a <= c.
;
;Input:
;   args: the list of elements to test.
;   slist: the association list representing the <= relation
;Returns:
;   T if the relation is transitive, else nil.
;
(defun transitivep (args slist)
(cond
 ((null args) t)
 ((not (trans2 (car args)
              (cadr (assoc (car args) slist)) slist)) nil)
 (T (transitivep (cdr args) slist))))

;TRANS2 determines if a <= b and b <= c means a <= c for a
;specific c in the logic.
;
;Input:
;   x1: the specific c
;   x1list: all values in the logic <= x1
;   slist: the <= relation
;Returns:
;   T if a <= b and b <= x1 implies a <= x1 for all a, b
;   in the logic. Otherwise nil.
;
(defun trans2 (x1 x1list slist)
(cond
 ((null x1list) t)
 ((not (sublist (cadr (assoc (car x1list) slist))
               (cadr (assoc x1 slist)))) nil)
 (t (trans2 x1 (cdr x1list) slist))))
```

```
;SUBLIST returns T if l1 is a sublist of l2, else nil.
;
;Input:
;   l1: the prospective sub-list
;   l2: the prospective super-list
;Returns:
;   T if every element of l1 is an element in l2, else nil.
;
(defun sublist (l1 l2)
(cond
 ((null l1) t)
 ((not (member (car l1) l2 :test equal)) nil)
 (t (sublist (cdr l1) l2))))

;REFLEXIVEP returns T if x <= x for all x in the logic.
;
;Input:
;   slist: the list representing the <= relation.
;Returns:
;   T if the relation is reflexive, else nil.
;
(defun reflexivep (slist)
(cond
 ((null slist) t)
 ((not (member (caar slist) (cadar slist) :test equal)) nil)
 (t (reflexivep (cdr slist)))))

;ANTISYMMETRICP determines if the relation is
;antisymmetric.
;
;Input:
;   args: the list of logic values to test
;   slist: the relation
;Returns:
;   T if the relation is antisymmetric, else nil.
;
(defun antisymmetricp (args slist)
(cond
 ((null args) t)
 ((not (symm2 (car args) (cadr (assoc (car args) slist))
                                slist)) nil)
 (t (antisymmetricp (cdr args) slist))))
```

```
;SYMM2 determines if x <= y and y <= x for a specific x.
;
;Input:
;   x1: the specific x
;   x1list: the list of all elements <= x1
;   slist: the relation
;Returns:
;   T if x <= y and y <= x implies x = y for all y in
;   the logic.
;
(defun symm2 (x1 x1list slist)
(cond
 ((null x1list) t)
 ((and (not (equal x1 (car x1list)))
       (member x1 (cadr (assoc (car x1list) slist))
                                           :test equal)) nil)
 (t (symm2 x1 (cdr x1list) slist))))

;FIND-IDENTS finds all the identity elements in the logic.
;Element e is an identity element if e ^ x = x ^ e = x' where
;x and x' are in the same cycle-class for all x in the logic.
;
;Input:
;   vals: the remaining logic values to check.
;   clist: a list of all the cycle-classes in the logic.
;Returns:
;   A list of all the identity elements in the logic. If there
;   are none, returns nil.
;
(defun find-idents (vals clist)
(cond
 ((null vals) nil)
 ((check1 (car vals) vlist clist)
  (cons (car vals) (find-idents (cdr vals) clist)))
 (t (find-idents (cdr vals) clist))))

;CHECK1 checks to see if arg is an identity element of the
;logic;
;
;Input:
;   arg: the element to be tested as an identity
;   arglist: all the values to check against
;   clist: a list of all the logic's cycle-classes
;Returns:
;   T if arg is an identity element, else nil.
;
(defun check1 (arg arglist clist)
(cond
 ((null arglist) t)
 ((not (member (car arglist)
               (find-cycle-class (^ (list arg (car arglist)))
                                 clist))) nil)
 (t (check1 arg (cdr arglist) clist))))

;FIND-CYCLE-CLASS finds the cycle-class of arg.
;
;Input:
;   arg: the element whose cycle-class is to be found.
;   clist: the list of all the logic's cycle-classes.
;Returns:
;   The cycle-class of arg, or nil if there is none.
;
(defun find-cycle-class (arg clist)
(cond
 ((null clist) nil)
 ((member arg (car clist)) (car clist))
 (t (find-cycle-class arg (cdr clist)))))
```

BIBLIOGRAPHY

Aristotle. Prior Analytics, trans. H. P. Cooke and H. Tredennick. London: William Heinemann Ltd., 1967.

Church, Alonzo. Introduction to Mathematical Logic. Priceton: Princeton University Press, 1956.

Copi, Irving M. Introduction to Logic. New York: Macmillan Publishing, 1972.

Grimaldi, Ralph P. Discrete and Combinatorial Mathematics. Reading: Addison-Wesley Publishing Co., 1985.

Haack, Susan. Deviant Logic. Cambridge: Cambridge University Press, 1974.

________. Philosophy of Logics. Cambridge: Cambridge University Press, 1978.

Hilbert D. and Ackermann, W. Principles of Mathematical Logic. New York: Chelsea Publishing Co., 1950.

Kneale, William and Kneale, Martha. The Development of Logic. Oxford: Clarendon Press, 1984.

Langer, Susan K. An Introduction To Symbolic Logic. New York: Dover Publications, Inc., 1953.

Lipschutz, Seymour. Discrete Mathematics. New York: McGraw-Hill, Inc., 1976.

Liu, C. L. Elements of Discrete Mathematics. New York: McGraw-Hill, Inc., 1977.

________. Introduction to Combinatorial Mathematics. New York: McGraw-Hill, Inc., 1968.

Moussavi, Massoud. Modeling Rule-Based Systems Using a Six Valued Logic. Research Proposal. George Washington University, 1986.

Post, E. L. "A General Theory of Elementary Propositions." The American Journal of Mathematics, xiii (1921), 163-185.

Rescher, Nicholas. Topics in Philosophical Logic. Dordrecht: D. Reidel, 1968.

________. Many-valued Logic. New York: McGraw-Hill, Inc. 1969.

Rosenbloom, Paul C. The Elements of Mathematical Logic. New York: Dover Publications Inc., 1950.

Ross, Kenneth A. and Wright, Charles R. B. Discrete Mathematics. Englewood Cliffs: Prentice-Hall, Inc., 1985.

Russell, B. and Whitehead, A. Principia Mathematica. Cambridge, 1910.

Stanat, Donald F. and McAllister, David F. Discrete Mathematics in Computer Science. Englewood Cliffs: Prentice-Hall, Inc., 1977

Stolyar, Abram Aronovich. Introduction to Elementary Mathematical Logic. Hanover: Halliday Lithograph Corp., 1970.

Turner, Raymond. Logics for Artificial Intelligence. West Sussex: Ellis Horwood Limited, 1984.

Winston, Patric H. Artificial Intelligence. Reading: Addison Wesley, 1984.